



Rapport de stage de fin d'études

ENSEIRB, I3 (PRCD)

Signatures numériques évoluées en XML
(XMLDSIG et XAdES)

Moez Ben MBarka

Encadré par :
Julien Stern (Docteur en cryptographie, Cryptolog).
Thomas Pornin (Docteur en cryptographie, Cryptolog).
Antoine Rollet (Docteur en Informatique, ENSEIRB).

23 septembre 2007

Remerciements

Je tiens à remercier toutes les personnes qui ont aidé à la réalisation de ce rapport. Je commence par citer tous mes professeurs à l´ENSEIRB et à l´université de Bordeaux 1. Je remercie en particulier Mr Mosbah qui m'a encouragé à faire ce stage dans ce domaine qui est la cryptographie.

Je remercie également, mes encadreurs dans l´entreprise Julien Stern et Thomas Pornin ainsi que tous les employés, développeurs et commerciaux, qui ont tout fait pour que le stage se déroule en bonnes conditions.

Résumé

Actuellement, de plus en plus d'ordinateurs sont interconnectés à Internet ou à des réseaux similaires. De tels réseaux ont la vocation de devenir la plate-forme universelle d'échange de produits et de services. La sécurisation des transactions à travers ces réseaux devient donc primordiale. La signature numérique est un des services de base de tout système d´information sécurisé. Elle est une composante fondamentale des transactions effectuées sur Internet. Basée sur des standards de la cryptographie à clé publique, elle offre l'équivalent numérique de la signature papier classique et elle peut garantir : l'intégrité, l'authentification et la non-répudiation. Avec la criticité croissante des échanges établis sur Internet, ainsi que les améliorations progressives des outils de sécurité, un cadre juridique vient compléter l'ensemble. La directive européenne 1999/93/CE, définit un format de signatures, dit "signatures évoluées" associant une valeur juridique et légale à la notion de la signature numérique.

D'autre part, XML se présente actuellement comme le support de données le plus populaire sur Internet. Par sa flexibilité et sa généricité, XML semble le moyen le plus adopté pour échanger des données entre des applications hétérogènes et distribuées. D'où le besoin de le doter d'outils nécessaires pour le rendre un moyen de transport de données aussi sûr qu'efficace. Dans ce cadre, un ensemble de standards et recommandations ont suivi pour définir des moyens permettant de sécuriser les échanges de messages XML. "XML-Signature Syntax and Processing (XMLDSIG)", une recommandation W3C*, est l'un des premiers de ces standards. Cette recommandation définit des règles pour la production et la validation des signatures XML. D'autres standards ont également suivi pour couvrir d'autres besoins en sécurité comme la confidentialité et la gestion des clés. Suite à la légalisation juridique des signatures numériques, "XML Advanced Electronic Signatures (XAdES)" a suivi pour permettre de produire des signatures "évoluées" en XML conformes à la directive européenne. Ce rapport introduit des notions liées à la signature numérique et étudie les deux formats de signatures en XML (XMLDSIG et XAdES) puis présente une implémentation en Java de ces deux standards.

Mots clé : cryptographie, infrastructure à clés publiques, signature numérique, XML.

*W3C : World Wide Web Consortium

Table des matières

Table des figures

Chapitre 1

Introduction

1.1 Rappel du sujet de stage

Le sujet du stage validé par l'école est : Étude et implémentation des standards XMLDSIG (XML-Signature Syntax and Processing) et XAdES (XML Advanced Electronic Signatures) pour les intégrer dans la suite dans des solutions de gestion de PKI (Public Key Infrastructure).

En plus de stage de PFE, le même sujet a servi comme stage de Master recherche*. Ce rapport reprend des parties du mémoire du Master† soutenu le 16 Juin en mettant plus en relief les aspects conception et implémentation.

Le stage s'est déroulé principalement en deux parties :

1. Lecture et compréhension du standard XMLDSIG[17] spécifiant les règles de production et validation des signatures numériques en format XML ainsi que l'extension XAdES[32] spécifiant un format XML pour les signatures évoluées conformes à la directive européenne 1999/93/CE[4].
2. Implémentation de ces standards en Java.

XMLDSIG[17] est devenu une recommandation W3C et de l'IETF‡ depuis 12 Février 2002, XAdES[32] est une spécification ETSI§ dont la dernière version a été publiée en Mars 2006. Ces deux documents représentent donc les références principales pour ce stage :
- XML-Signature Syntax and Processing (RFC 3275).
- XML ADvanced Electronic Signatures (ETSI TS 101 903).

D'autre part, la compréhension de ces deux standards exige des acquis dans le domaine de la cryptographie et particulièrement la cryptographie à clé publique. Au cours de ce stage, j'ai été donc amené à la lecture partielle ou complète des spécifications de plusieurs standards concernant les signatures numériques, les certificats et plus généralement la gestion d'une infrastructure à clé publique (PKI ou Public Key Infrastructure[15]). Ces documents seront cités dans la suite de ce rapport.

*Master recherche à l'université de Bordeaux 1 (option SDRP)

†Une copie complète de ce mémoire peut être téléchargé ici : http ://uuu.enseirb.fr/ benmbark/sources/paper_fr.pdf

‡IETF : The Internet Engineering Task Force.

§ETSI : European Telecommunications Standards Institute

1.2 Problématiques, objectifs et résultats

1.2.1 Contexte général

Les réseaux informatiques ouverts tels que l'Internet ont été techniquement optimisés pour assurer le transport de données. Dans cette optique les aspects liés à la sécurité n'étaient pas une priorité pour les protocoles tel que IP¶[12]. Or, Internet ayant vocation à devenir la plate-forme universelle d'échange de produits et de services, la sécurité devient primordiale. Le développement de la cryptographie avec les progrès conjugués des mathématiques et de l'informatique ont permis, depuis les années 1970, de disposer progressivement d'une suite complète de solutions pour assurer les aspects les plus importants de la sécurité dans un réseau tels que l'Internet, à savoir : l'authentification, la confidentialité et l'intégrité. La signature numérique constitue un service de base de cet ensemble de solutions. Dite aussi signature électronique, c'est un procédé cryptographique permettant de garantir l'intégrité du message signé et l'authenticité de son expéditeur. Elle peut également garantir la non-répudiation, c'est-à-dire empêcher l'expéditeur de nier avoir signé le message.

D'autre part, XML[34], est devenu actuellement l'un des plus importants supports de données sur Internet. Il s'est imposé comme moyen générique et flexible d'échange de données entre des entités hétérogènes et distribuées. XML dispose également d'un ensemble d'outils et technologies facilitant la manipulation et le transport du contenu numérique. Dans ce cadre, tout un ensemble de standards basés sur XML ont été définis pour lui associer une sécurité primordiale pour tout langage de transport de données sur Internet digne de ce nom. "XML-Signature Syntax and Processing"[17], une recommandation définie conjointement par l'IETF et W3C, est le standard de base permettant de décrire en XML une signature numérique ayant toutes les propriétés d'une signature numérique classique (à savoir, l'intégrité et l'authentification).

Avec la criticité croissante des transactions et échanges établis sur Internet, ainsi que les améliorations progressives des outils de sécurité, un cadre juridique adapté à la notion numérique et "immatérielle" de la signature a suivi le développement de l'Internet. La directive européenne 1999/93/CE[4] définit dans ce cadre un format de signature dit "Signature évoluée" qui associe à un document numérique signé un caractère légal et juridique. "XML Advanced Electronic Signature (XAdES)"[32], une spécification technique de l'ETSI, définit un format de signature basé sur XMLDSIG[17], et conforme aux modalités de la directive européenne.

1.2.2 Objectifs et résultats

Objectifs :

Les objectifs de ce stage ont été fixés comme suit :

1. Faire une étude approfondie des standards XMLDSIG et XAdES en particulier et les standards de sécurité XML en général. Cela inclue aussi la compréhension de CMS et les principaux standards de la cryptographie à clé publique (certificat, CRL‖, OCSP**, gestion de clés...).

2. Réaliser une implémentation de base de XMLDSIG.

3. Ajouter une couche XAdES minimale permettant de produire et valider des signatures évoluées conformément à la directive européenne.

¶IP : Internet Protocol

‖CRL : Certificate Revocation List

**OCSP : Online Certificate Status Protocol

Résultats :

Les objectifs cités ci-dessus ont été bien menés dans les délais. En plus de XMLDSIG et XADES, j'ai pu participé au développement d'autres bibliothèques. Je cite :
- une bibliothèque pour l'encodage/décodage en XML de paramètres cryptographique (algorithmes, clés...).
- une bibliothèque pour l'encodage/décodage en XML des politiques de validation des signatures conformément à une spécification technique de l'ETSI[8].
- une API générique pour la production des signatures en XML et CMS[††][19].
- Une implémentation des structures des protocoles définies par le standard DSS[‡‡][6] de OASIS[§§]. J'ai également participé au développement des composants d'un serveur de signature utilisant ces protocoles. Une présentation de DSS ainsi qu'une réflexion sur l'architecture générale du serveur sont en annexe de ce rapport.

1.3 Organisation du document

La suite de ce rapport est organisée de la façon suivante :
- **chpitre 2 :** présente l'entreprise dans laquelle s'est effectué le stage.
- **chapitre 3 :** constitue un préliminaire nécessaire pour la compréhension des chapitres suivants. Il donne une introduction rapide à des concepts importants de la cryptographie et en particulier la cryptographie à clé publique. Il donne également un aperçu sur XML et les outils liés qui seront utilisés dans la suite.
- **chapitre 4 :** détaille le standard XMLDSIG[17] : processus de production, validation et règles syntaxiques.
- **chapitre 5 :** détaille l'extension XAdES[32].
- **chapitre 6 :** présente les implémentations de XMLDSIG et XAdES.
- **chapitre 7 :** ce dernier chapitre donne les résultats des tests d'intéropérabilités effectués à la fin du stage.

Plusieurs annexes sont joints à ce rapport :
- Un exemple complet d'une signature XAdES.
- Le schémas des signatures XML.
- Structure générale d'une signature XAdES.
- Le cahier des charges.
- Un manuel d'utilisation de XMLDSIG rédigé en anglais sur le wiki de l'entreprise.
- Une réflexion sur l'architecture générale d'un serveur de signature DSS rédigé également en anglais sur le wiki de l'entreprise.

[††]CMS : Cryptographic Message Syntax
[‡‡]DSS : Digital Signature Services (http ://www.oasis-open.org/committees/tc_home.php ?wg_abbrev=dss).
[§§]OASIS : Organization for the Advancement of Structured Information Standards (http ://www.oasis-open.org).

Chapitre 2

Présentation de l'entreprise

L'entreprise dans laquelle s'est déroulé le stage est : *Cryptolog International*. Cryptolog est une jeune entreprise spécialisée en cryptographie et en particulier la gestion des identités numériques. Le stage s'est déroulé dans le pôle R&D de l'entreprise constitué par 2 doctorant en cryptographie, 3 ingénieurs et 4 stagiaires. Le stage a été encadré par Julien Stern et Thomas Pornin, docteurs en cryptographie.

Cryptolog a été fondé en 2001 par trois associés qui forment encore le bureau directive de l'entreprise :

- **Alexandre Stern**(président) : Avant de fonder Cryptolog, Alexandre a travaillé auprès du Directeur de la Stratégie du groupe CS Communications & Systems. Il est diplômé d'HEC, de la Leonard N. Stern School of Business, New York University et de l'Université Paris Dauphine.
- **Julien Stern**(directeur R&D) : Avant de fonder Cryptolog, Julien a effectué des recherches en cryptographie pour plusieurs grandes sociétés, parmi lesquelles IBM, AT&T, Intertrust et Lucent Technologies. Il est titulaire d'un Doctorat en cryptographie obtenu à l'ENS Lyon.
- **Thomas Pornin**(directeur technique) : Avant de fonder Cryptolog, Thomas a travaillé pour Gemplus et s'est dédié à la recherche fondamentale dans le domaine de la cryptographie symétrique. Il a publié de nombreux articles sur ce sujet, et est titulaire d'un Doctorat en cryptographie obtenu à l'ENS Paris.

2.1 Secteurs d'activité

Le terme "marché de la cryptographie" est généralement utilisé pour désigner le domaine d'activité des entreprises qui proposent des solutions complètes en cryptographie destinées à des utilisateurs qui ne connaissent pas nécessairement ce domaine. C'est le domaine d'activité de l'entreprise depuis sa création. Cryptolog dispose actuellement d'un ensemble de solutions complètes qui peuvent répondre aux plus important des besoins en matière de gestion des identités et de certification numérique. Le pôle R&D de l'entreprise participe également à plusieurs projets de recherche français et européens.

2.2 Produits Cryptolog

Les produits Cryptolog couvrent principalement les besoins suivants :

- **Authentification forte** : L'utilisation d'un système d'authentification forte à la place d'un système d'authentification traditionnel permet d'obtenir des preuves formelles de l'identité de toute personne accédant à des données sensibles, contribuant ainsi à réduire la fraude et les usurpations d'identité. L'authentification forte est basée sur l'utilisation en plus des paramètres d'identification traditionnels (comme un couple identifiant/mot de passe) de données d'authentification supplémentaires comme des certificats numériques.

- **Signature électronique** : Avec la tendance de dématérialisation des procédures qui mène vers la substitution du papier comme support des contrats et factures par des supports numériques, la sécurité qui porte sur l'authentification et l'intégrité des ces transactions devient primordiale. La signature électronique est un des procédés de base qui permet de garantir la sécurité (authentification et intégrité) des transactions numériques en vue de leur donner des valeurs légales.

- **Messagerie sécurisée** : Dans le même cadre la messagerie sur internet représente de plus en plus un des moyens de communication le plus utilisé. Le chiffrement des messages échangés permet de garantir leur confidentialité et intégrité.

Parmi les principaux produits Cryptolog (voir figure 2.1) on peut citer :

- **Cryptolog Identity** : une IGC*, le noyau central de tout projet de gestion des identités numériques.

- **Cryptolog Unicity** : un système innovant de stockage centralisé des clés privées vous permettant de déployer une solution de gestion des identités numériques sans nécessiter de support physique (carte à puce/token USB) pour chaque utilisateur.

- **Cryptolog WebPass** : une applet d'authentification et de certification pour tout navigateur web.

- **Cryptolog Unicity plug-in** : l'équivalent logiciel d'un lecteur de carte à puce.

- **Cryptolog Eternity** : un serveur d'horodatage apportant la preuve qu'une action a bien été effectuée à un moment précis.

- **Cryptolog Universal** : Token Interface (CUTE), un outil de gestion des certificats numériques agissant comme interface entre n'importe quelle IGC et tout token physique ou logiciel.

- **Cryptolog OCSP responder** : un outil de vérification en temps réel de la validité des certificats.

2.3 Environnement technique

Tous les développements ont été effectué avec Java dans un environnement de développement classique : poste de travail Linux, emacs comme éditeur de sources et cvs pour la gestion des versions.

L'entreprise utilise pour la gestion des bibliothèques java l'outil Maven†. Cet outil permet, en se basant sur des fichiers de description en XML, de compiler les projets et générer la documentation

*IGC : Infrastructure de Gestion de clés.

†Apache Maven : http ://maven.apache.org/

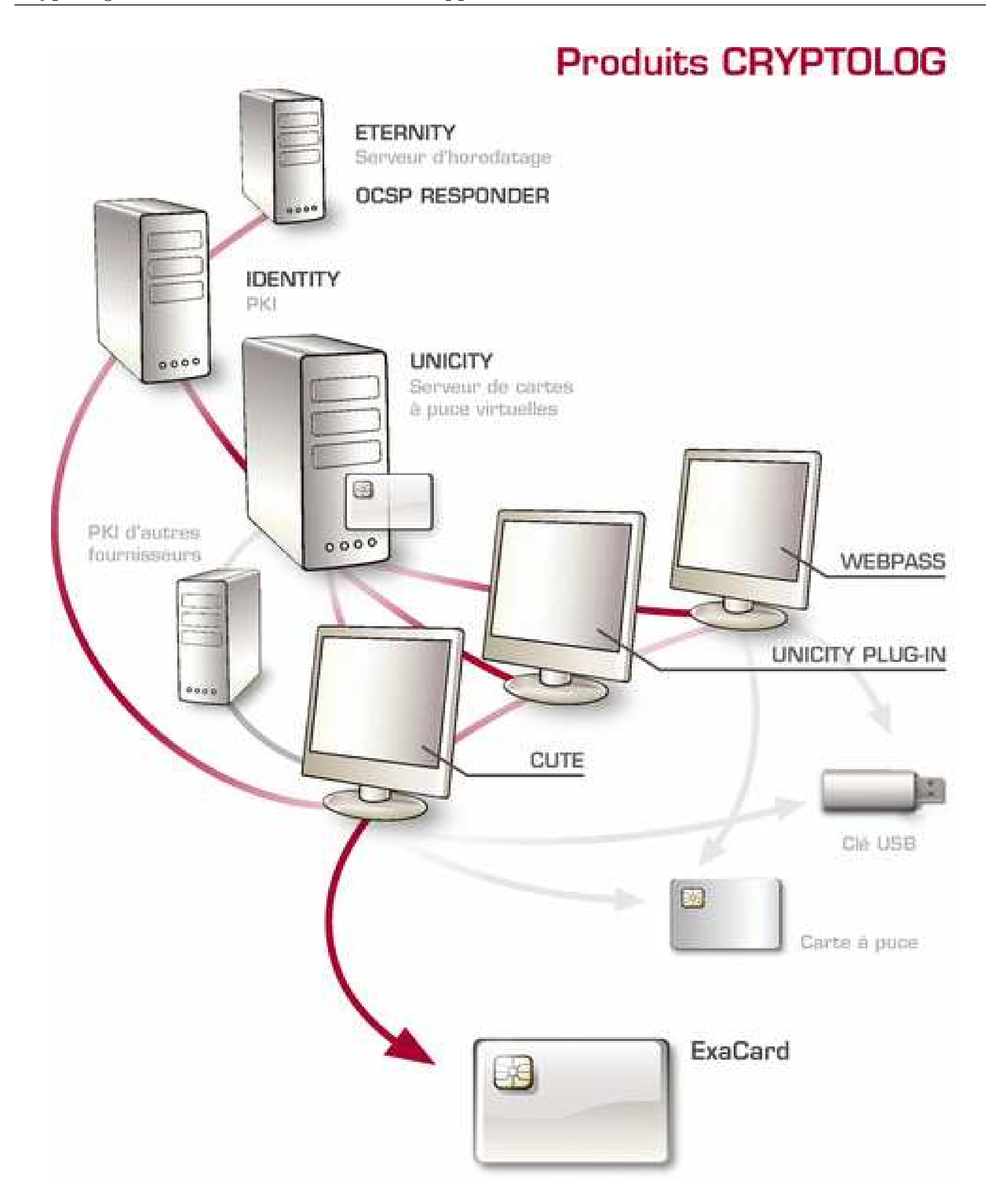


Fig. 2.1 – Les produits Cryptolog.

ainsi que plusieurs rapports permettant la maintenance du code source.

D'autre part, l'entreprise utilise le wiki Confluence‡ comme moyen de documentation et travail collaboratif. Le développement des différents bibliothèques de ce stage, a été précédé par l'écriture

‡Confluence : http ://www.atlassian.com/software/confluence/

d'une documentation en anglais sur cet outil. Des manuels d'utilisation ont été aussi rédigé sur le wiki à la fin du stage. Certains de ces manuels sont joints en annexe à ce rapport.

Chapitre 3

Préliminaires

Dans ce chapitre, on va présenter des notions de base nécessaires pour la compréhension de la suite du rapport. La première section introduit des notions de base de la cryptographie à clé publique. La section suivante présente XML et quelques technologies liés qui seront utilisées dans la suite. Enfin, la dernière section introduit différents formats de signatures qui seront détaillés dans les chapitres suivants.

3.1 Introduction à la cryptographie

Aujourd'hui de plus en plus de transactions sensibles se font sur Internet ou des réseaux similaires. Ces transactions peuvent passer par un certains nombre de serveurs intermédiaires.Cela rend possible pour des entités "malicieuses" d'intervenir sur les données échangées de plusieurs façon différentes, a savoir :

- Écoute clandestine (ou "Eavesdropping") : les données ne sont pas altérées mais leur confidentialité est compromise.
- Altération des données (ou "Tampering") : les données originales sont interceptées et modifiées puis envoyées au destinataire initial.
- Usurpation d'identité (ou "Impersonation" : les données ne sont pas envoyées par la personne que le destinataire croit être l'expéditeur du message.

Dépendant de la nature et l'importance des transactions, ces attaques peuvent être fatale aussi bien pour les particuliers que pour les entreprises et institutions. La cryptographie offre un ensemble de techniques et standards pour se protéger contre ces différentes attaques. Un système de transactions sûr doit être doté des techniques de protection suivantes :

- Chiffrement et déchiffrement : permettent d'assurer la confidentialité des données. Même, si la transaction est interceptée, la confidentialité n'est pas compromise vue que les données ne transitent pas en clair.
- Vérification de l'intégration des données : Le destinataire doit pouvoir vérifier que les données reçues sont exactement les données envoyées sans aucune altération.
- Authentification : Permet au destinataire de s'assurer de l'identité de son correspondant.
- La non-répudiation : Empêche le correspondant de nier dans le futur d'avoir envoyé le message.

La cryptographie à clé publique est composée d'un ensemble de standards facilitant la mise en oeuvre de ces différents techniques. La section suivante introduit d'abord la cryptographie à clé privé.

3.1.1 Cryptographie à clé privé

La cryptographie à clé privé (dite aussi cryptographie symétrique) repose sur l'utilisation de la même clé, dite clé privé ou secrète, à la fois pour le chiffrement et le déchiffrement (voir figure 3.1). Elle permet d'assurer la confidentialité des données ainsi que leur authentification du faite que seules les personnes possédant la clé peuvent chiffrer et déchiffrer un message donné. Cependant, ces propriétés ne restent vrai que tant que la clé n'a pas été divulgué à un tiers. Si une entité malicieuse accède à cette clé, elle peut, pas seulement déchiffrer les messages échangés, mais aussi en envoyer en se faisant passer comme une des entités initiales ayant accès à la clé.
D'autre part, la cryptographie symétrique devient de plus en plus difficile à mettre en oeuvre quand le nombre d'utilisateurs voulant communiquer de façon sûr augmente. En effet, il faut au moins une clé privé pour tout couple d'utilisateurs ; ce qui pose des problèmes de gestion de clés.
Un autre problème fondamental pour la cryptographie symétrique est l'échange de la clé secrète entre les deux parties communiquantes. La confidentialité de cet échange est cruciale pour toute les communications suivantes.

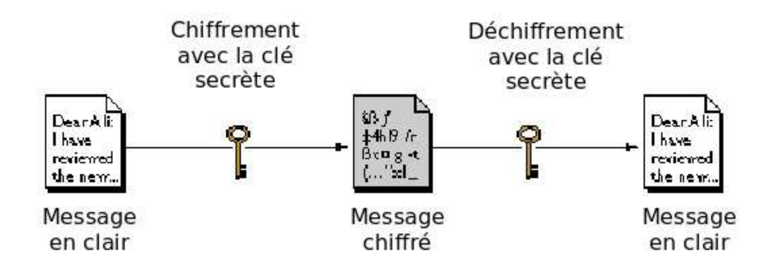


Fig. 3.1 – Chiffrement/déchiffrement à clé secrète.

La cryptographie à clé publique (ou cryptographie asymétrique) à été mise au point peur résoudre en parties les problèmes de gestion de clés de la cryptographie symétrique. Elle peut aussi être utilisé comme solution pour le problème de l'échange de la clé secrète.

3.1.2 Cryptographie à clé publique

La cryptographie à clé publique repose sur l'utilisation de deux clés différentes pour le chiffrement et le déchiffrement (voir figure 3.2). L'une est dite clé privé et elle doit rester secrète. La deuxième clé, dite clé publique, peut être librement publiée. Quand deux parties souhaitent échanger un message de façon sûre, le premier utilise la clé publique (qui peut être récupérée par exemple à partir d'un annuaire de clés) du destinataire pour chiffrer le message. Le correspondant utilise sa clé privé (supposée connue uniquement par lui) pour déchiffrer le message.

La cryptographie asymétrique est basé sur une fonction facile à calculer dans un sens (appelée fonction à sens unique) et mathématiquement très difficile à inverser sans une information supplémentaire qui est la clé privée.
Ce système permet donc d'échanger des messages chiffrés sans avoir à partager une information secrète ce qui évite le problème d'échange de clés évoqué par la cryptographie symétrique.

Plusieurs standards et protocoles ont apparus autour de la cryptographie à clé publique. Les sections suivantes introduisent certains de ces standards pour présenter les notions de certificat et CRL (Certificate Revocation List) avant de détailler une application importante de la cryptographie asymétrique qui est la signature numérique.

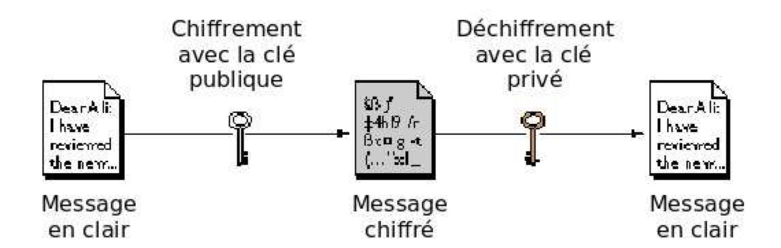


Fig. 3.2 – Chiffrement/déchiffrement à clé publique.

Les certificats numériques

La cryptographie asymétrique est basée sur la publication et le partage des clé publiques. Ce partage peut s'effectuer à travers des annuaires LDAP[14] (Lightweight Directory Access Protocol). Cela rend l'annuaire une cible idéale pour des attaques d'usurpation d'identité. Une personne "malicieuse" peut simplement remplacer une clé publique dans l'annuaire par sa propre clé publique pour se faire passer pour une autre personne. Ainsi, il sera capable de déchiffrer tous les messages qui vont être chiffrer avec la "mauvaise" clé publique à l'intention de la personne possédant la clé publique initiale. La notion de certificat est apparue pour remédier à ce problème. Un certificat permet d'identifier une personne ou une entité en lui associant une clé publique. Il peut être vue comme l'équivalent numérique d'une carte d'identité classique.

Lorsqu'un utilisateur désire communiquer avec une autre personne, il lui suffit de se procurer le certificat du destinataire. Ce certificat contient le nom du destinataire, ainsi que sa clé publique et est signé par l'autorité de certification qui a délivré le certificat.

Les CRL (Certificate Revocation List ou Liste de Certificats Révoqués)

Comme une carte d'identité standard ou un permis de conduite, un certificat peut à tout moment être suspendu par l'autorité qui l'a émis même s'il est encore dans sa période de validité. Par exemple, si un employé est renvoyé, son certificat doit être révoqué pour empêcher toute utilisation malicieuse. La révocation des certificats peut être géré de plusieurs manières. Une approche largement utilisée est la publication périodique de la liste des des numéros de série des certificats à invalider. Cette liste (dite CRL) est signée avec la clé privé de l'autorité émettrice et peut être publiée via le même annuaire de publication des certificats. A chaque requête de validation, l'administrateur doit vérifier que le certificat n'est pas cité dans aucune CRL.

Les signatures numériques

Le chiffrement permet d'assurer la confidentialité des données et donc d'éviter les attaques de type "écoute clandestine" (voir 3.1). Cependant, il ne protège pas les données contre l'altération et l'usurpation d'identité.

L'intégrité des données peut être assurée par des fonctions cryptographiques appelées fonctions de hachage (hash functions). Une fonction de hachage est une fonction permettant de calculer à partir d'une entrée initiale une valeur (appelé empreinte numérique ou haché) de taille plus petite. Un algorithme de hachage doit avoir les deux propriétés suivantes :

- A partir de la valeur de l'empreinte, il est impossible de revenir à l'entrée initiale.

- Pour une entrée donnée, il est impossible de trouver une autre entrée qui aurai la même empreinte numérique.

Les algorithmes de hachages les plus utilisés sont MD5 (Message Digest 5) et SHA1 (Secure Hash Algorithm).

Pour vérifier l'intégrité d'un message, le récepteur calcule de nouveau la valeur du haché et la compare avec celle reçu avec le message. Si les deux valeurs sont égales, il est peu probable que le message ait été altéré. Cependant, rien ne prouve que le message a bien été envoyé par celui qu'on croit être l'expéditeur. Pour remédier à ce problème, l'expéditeur chiffre la valeur du haché avec sa clé privée. La valeur obtenue s'appelle la signature numérique. Le récepteur utilise la clé publique de l'expéditeur pour déchiffrer la signature et retrouver la valeur de l'empreinte. Puis applique l'algorithme de hachage sur le message initiale et compare les deux empreintes ainsi obtenues. Si les deux valeurs sont égales, le récepteur peut être sûr des deux propriétés suivantes :

1. Le message reçu est exactement le message envoyé sans aucune altération vue que son haché est correcte.
2. Le message a bien été envoyé par la personne supposé être l'expéditeur vue qu'il a été chiffré par la clé privé associée à la clé publique de cette personne.

Le schémas de la figure 3.3 illustre les processus de production et vérification d'une signature numérique.

3.2 Introduction à XML

XML[34] (Extensible Markup Language) est un langage extensible de structuration des données. Il a été mis au point par le XML Working Group sous l'égide du World Wide Web Consortium (W3C) depuis 1996. A l'origine il a été inspiré du langage SGML (Standard Generalized Markup Language) en le rendant plus simple et utilisable sur le web. Un document XML est formé par un ensemble d'éléments définis par des balises et des attributs. Son extensibilité vient du faite qu'aucune limite ou contrainte n'est appliqué sur les noms ou la structure de ces éléments. XML peut être considéré comme un méta langage permettant de définir d'autre langages.

Le vocabulaire suivant va être utilisé dans les chapitres suivants :
- Élément (ou noeud) : Toute balise définit un élément. Un élément peut avoir un ou plusieurs attributs et peut être le parent d'un ou plusieurs éléments.
- Racine : Tout document XML possède une racine. C'est le premier élément qui contient tous les autres éléments du document.
- Attribut : Un attribut est définit par un couple nom-valeur et qui qualifie l'élément dans lequel il est définit.

On peut facilement comparer un document XML à un document HTML. Cependant il y a deux grandes différences entre les deux formats. D'une part, HTML est un langage figé (à nombre de balises limité) alors que XML est extensible ce qui permet de définir des nouvelle balises. D'autre part, HTML définit à la fois le contenu et la présentation, alors que XML peut être utilisé comme un support générique de données permettant de séparer le contenu de la forme. Ce qui permet par exemple de manipuler le même contenu par des applications différentes.

Depuis avoir été reconnu comme une recommandation du W3C, XML est devenu un des supports les plus utilisés sur le web. Plusieurs technologies et outils ont suivis pour faciliter la manipulation des documents XML. Les sections suivantes présentent certains de ces outils qui seront utilisés dans la suite.

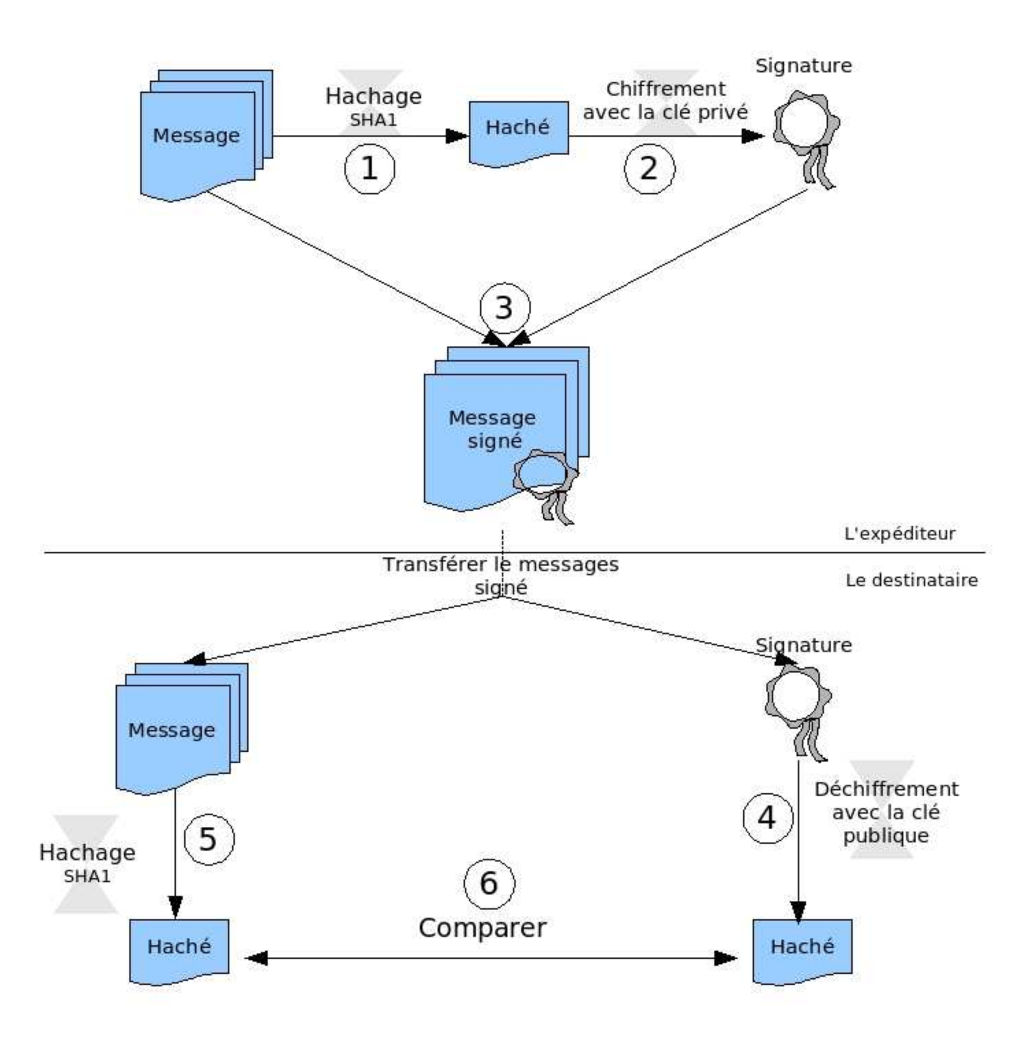


FIG. 3.3 – Production d'une signature numérique.

3.2.1 Outils XML :

Espaces de noms et schémas XML

Un des points fort de XML est son extensibilité qui offre une liberté totale sur les noms et la structure des éléments d'un document. Cependant, cette liberté peut engendrer des ambiguïtés pour les applications échangeant des données XML, du faite que la structure de ces données peut être aléatoire. Il est donc nécessaire de disposer d'un moyen permettant de fournir à l'application un schémas jouant le rôle d'un contrat qui spécifie à l'avance les structures que l'application est censée pouvoir traiter. Le standard XML-Schema[37] définit par le W3C permet de définir de façon non ambigu la structure de tout document XML. Un scémas XML définit la structure du document en spécifiant les noms des balises et les attributs ainsi que la structuration du document. XML-Schema[37] définit une multitude de fonctionnalités assez complexes pour une description totalement non ambigu d'un document. L'annexe 7.3.2 donne par exemple le schémas d'une signature XML.

D'autre part, un schémas XML utilise la notion de vocabulaire en associant à un document

XML un espace de nom définit par une URI[20] (Uniform Resource Identifier). La notion d'espace de noms ou vocabulaire a été introduite pour permettre de mélanger des contenus XML issus de contextes différents sans risque de conflit entre les balises.

XPath et XPointer :

XPath[40] (XML Path Language) est un langage déclarative permettant de spécifier des chemins dans un arbre représentant un document XML. XPath est aussi une recommandation de W3C.

Les expressions XPath sont évaluées par rapport à un noeud initial représentant le contexte de l'évaluation. Ces expressions permettent de sélectionner des noeuds ou des attributs atteignable à partir du contexte selon des chemins dans un arbre.

Toute expression est de la forme :

```
axe1::filtre1[prédicat1]/axe2::filtre2[prédicat2]
```

où *axe1* et *axe2* permettent de spécifier le chemin de recherche dans l'arbre. Les filtres et les prédicats sont des expressions dont l'évaluation permet de sélectionner les noeuds désirés dans le chemin.

XPointer[41] (XML Pointer Language) est une extension de XPath définit également par W3C. L'objectif de XPointer est de localiser un fragment d'un document XML local ou distant.

3.3 Formats de signatures numériques

Quand un destinataire reçoit la valeur d'une signature il a besoin de plus d'information pour vérifier sa validité. A savoir : le message initial, la clé publique associée à la clé privée qui a été utilisée pour le chiffrement ainsi que les algorithmes de hachage et chiffrement utilisés. D'où le besoin de pouvoir envoyer toutes ces information en même temps que la signature. Une solution "naïve" est de "packager" tout ça dans un format plus au moins arbitraire comme dans l'exemple suivant où le message est inséré sur la première ligne, suivi des identifiants des algorithmes utilisés puis de l'identifiant du signataire qui permettra de trouver sa clé publique et enfin la valeur de la signature.

```
Message signé
Ceci est un message important.
RSA-SHA1
Mr Dupont
MI6rfG4XwOjzIpeDDDZB2B2G8FcBYbeYvxMrO/
Ta7nm5ShQ26KxK71Ch+4wHCMyxEkBxx2HPO/7J
tPiZTwCVEZ1F5J4vHtFTCVB8X5eEP8nmi3ksdT
Q+zMtKjQII9AbCNxdA6ZtXfaOV4euO7UtRHyK1
7Exbd9PNFxnq46b/f8I=
```

Ce packaging peut être satisfaisant dans le cas où il y a toujours un seul destinataire et qui est informé au préalable du format utilisé. Cependant, dès qu'il y a un grand nombre de destinataires un tel packaging "brut" devient rapidement inutilisable du faite qu'il manque d'une structure bien définie. D'où la nécessité de formats de signature structurés et permettant de transporter les informations nécessaires à la validation. Les deux formats les plus connus sont les suivants :

- **CMS[19] (Cryptographic Message Syntax)** : CMS est un standard de l'IETF qui définit une syntax permettant de signer ou chiffrer tout contenu numérique. La syntaxe de CMS est basé sur le format ASN.1[1] (Abstract Syntax Notation One). ASN.1 étant une notation formelle permettant de décrire de façon abstraite des données structurées.
- **XMLDSIG[17] (XML Digital Signature)** : XMLDSIG est définit par le W3C et l'IETF. Il utilise XML comme support pour transporter l'ensemble des donnée formant la signature. De ce faite, il bénéficie de toute la flexibilité qu'offre XML en terme de structuration de données. Le chapitre suivant détaillera la syntaxe ainsi que les processus de génération et validation des signatures XML.

Chapitre 4

XMLDSIG : XML Digital SIGnature

4.1 Introduction

Le standard XMLDSIG[17] est le résultat du travail du groupe « XML Signature WG » qui est un groupe conjoint du W3C et de l´IETF. De ces travaux sont issus une « Recommandation » du W3C et un « Standard track » de l'IETF, « XML-Signature Syntax and Processing» (RFC 3275). La recommandation W3C définit les règles syntaxiques ainsi que les processus de génération et validation des signatures XML.

Étant basé sur XML, XMLDSIG bénéficie de plusieurs technologies liées à ce support de données comme XPath, XPointer et XSLT (voir la section 3.2). La combinaison de ces outils offre une grande flexibilité à XMLDSIG permettant par exemple de transformer un document avant de le signer ou de signer seulement certaines parties d'un document donnée. Ces mécanismes seront détaillés dans les sections qui suivent.

Les signatures XML peuvent être représentées sous une vue structurelle et une vue fonctionnelle. La première section de ce chapitre présente la vue fonctionnelle en détaillant les processus de production et validation d'une signature XML. La section suivante présente la vue structurelle en détaillant les structures qui composent ces signatures.

4.2 Vue fonctionnelle : processus de génération et validation

Produire une signature XML revient à produire un document XML selon des règles syntaxiques définit par la recommandation W3C[17]. Une signature XML a la structure simplifiée suivante :

```
<Signature>
 <SignedInfo>
 </SignedInfo>
 ...
 <SignatureValue/>
 ...
 <KeyInfo>
 </KeyInfo>
```

```
</Signature>
```

La signature comporte donc trois parties principales :
- **SignedInfo** : contient le contenu effectivement couvert par la signature.
- **SignatureValue** : La valeur de la signature.
- **KeyInfo** : contient les information cryptographiques nécessaires à la validation de la signature.

De façon simple, La production consiste à produire l'élément *SignatureValue* en appliquant l'algorithme de la signature sur l'élément *SignedInfo*. La validation consiste à vérifier cette valeur de la signature en utilisant les information transporté dans *KeyInfo*. Les sections suivantes détaillent les processus de génération et validation ainsi que les types de signatures XML.

4.2.1 Production des signatures XML

La première étape consiste à former l'élément *SignedInfo*. Cet élément à la structure simplifiée suivante (où '+' indique que l'élément peut apparaître une ou plusieurs fois et '?' indique que l'attribut peut apparaître au plus une fois.)

```
<SignedInfo>
 <CanonicalizationMethod />
 <SignatureMethod />
 (<Reference URI? >)+
</SignedInfo>
```

XMLDSIG permet de signer tout type de contenu numérique. Le contenu à signer est adressé avec un ou plusieurs éléments *Reference*. Ceci permet à une seule signature XML de signer plusieurs ressources distribuées. Chaque ressource est référencé en utilisant la syntaxe des URI[20]. La section suivante détaille les mécanismes utilisés par les élément *Reference*.

Adresser les ressources signées :

La syntaxe des URI[20] offre un moyen simple et extensible pour référencer tout type de ressources locales ou distantes. Dans le cas ou le document à signer est en XML, la combinaison de cette syntaxe avec Xpointer[41] (voir 3.2) permet même de référencer des fragments spécifiques dans le document à signer. Prenons comme exemple les deux éléments *Reference* suivants :

Exemple d'éléments Reference

```
<SignedInfo>
...
<Reference URI=''http://www.test.com/document.xml''>
...
</Reference>
...
<Reference URI=''#toSign''>
...
</Reference>
...
</SignedInfo>
....
....
```

```
14 <Important Id=''toSign''>
15 Sign me...
16 </Important>
17 ...
18
```

La première Référence (ligne 3) avec l'URI : "http ://www.test.com/document.xml#toSign" indique que le premier contenu signé est le fichier distant *document.xml* hébergé sur le serveur du site *www.test.com*. La deuxième référence (ligne 7) avec l'URI plus courte : "#toSign" référence un contenu local situé dans le même document XML que la signature. Ce contenu est identifié grâce à son attribut *Id* (lignes de 14 à 16).

Regardons maintenant de plus près l'élément *Reference*. Sa structure simplifiée est la suivante (où '+' indique que l'élément peut apparaître une ou plusieurs fois, '?' indique que l'élément peut apparaître au plus une fois.) :

```
<Reference URI=''...''>
 (<Transforms>
   (<Transform/>)+
  </Transforms>)?
  <DigestMethod>...</DigestMethod>
  <DigestValue>...</DigestValue>
</Reference>
```

Il peut être nécessaire pour certaines applications de transformer le contenu désigner par l'URI avant d'appliquer la signature. En général, un utilisateur signe un contenu tels qu'il le voit. Les transformations peuvent être utilisés pour appliquer des filtres sur un contenu initial. Les éléments *Transform* permettent de définir une séquence d'algorithmes de transformations à appliquer sur le contenu référencé par l'URI. Ensuite, l'algorithme de hachage définit par l'élément *DigestMethod* est utilisé pour hacher le résultat des transformations. La valeur du haché ainsi obtenue, définit ensuite l'élément *DigestValue*.

Assembler l'élément *SignedInfo*

En plus des éléments *Reference*, le *SignedInfo* contient deux autres éléments qui sont : *SignatureMethod* et *CanonicalizationMethod*. Le premier désigne l'algorithme de signature qui va être utilisé. Le deuxième désigne l'algorithme de canonicalization qui va être utiliser pour "régulariser" le contenu du *SignedInfo* avant de calculer la signature.

Résumons donc les étapes de l'assemblage du *SignedInfo* :

1. Pour chaque resource à signer :
   (a) former un élément Reference avec l'URI désignant la ressource à signer et l'ensemble des transformations à appliquer avant le calcul du haché.
   (b) Appliquer les transformations en séquence : l'entrée de la transformation n est la sortie de la transformation $n-1$ ou le contenu désigné par l'URI pour $n = 1$.
   (c) Calculer la valeur du haché en appliquant l'algorithme du hachage choisit sur la sortie de la dernière transformation.

   (d) Ajouter à la *Reference* les élément *DigestMethod* et *DigestValue* désignant respectivement l'algorithme de hachage et la valeur du haché.

2. Concaténer à la série des éléments *Reference* ainsi obtenues les éléments *SignatureMethod* et *CanonicalizationMethod*.

Normaliser l'élément SignedInfo

La valeur de la signature est obtenu en appliquant l'algorithme de signature désigné par l'élément *SignatureMethod* sur la représentation XML de l'élément *SignedInfo*. La signature protège donc tout le contenu de cet élément. A noter donc qu'une signature XML ne couvre pas directement le contenu signé mais elle couvre une structure formée par la valeur de l'empreinte numérique résultant de l'application d'un algorithme de hachage sur ce contenu.

Pour que la signature reste valide, le signataire et le vérificateur doivent manipuler exactement la même représentation XML de l'élément *SignedInfo*. Cependant, XML , par son format libéral, permet d'écrire le même contenu XML de plusieurs façons différentes. Même des modifications mineures non significatives pour l'interprétation du document XML conduisent à des problèmes de validation vu que tout algorithme de hachage est sensible à la modification du moindre caractère. Par exemple, en appliquant l'algorithme de hachage SHA1 à la structure suivante faisant partie du *SignedInfo* :

```
<Reference URI=''http://www.test.com/document.xml''>
```

on obtient la valeur suivante :

```
b5525059d22848b9bb8b9807e80f499df227efa8
```

Maintenant en appliquant le même algorihtme à la structure suivante déduite de la précédente en ajoutant simplement quelque espaces avant l'attribut "URI" :

```
<Reference        URI=''http://www.test.com/document.xml''>
```

on obtient évidemment un haché totalement différent :

```
8aee97ee42fef65b97ec0c2a1f44d8513e96fc2d
```

Le nombre d'espaces entre le nom de la balise et l'attribut est totalement non significatif pour l'interprétation du document XML et dépend de l'implémentation utilisé pour parser le document. Il peut conduire cependant à l'invalidation de la signature. Pour remédier à ce type de problèmes, XMLDSIG prévoit des algorithmes de canonicalization permettant de normaliser un contenu XML le rendant indépendant de l'implémentation utilisée.

La canonicalization XML

Le processus de canonicalization permet de normaliser un contenu XML. Il est nécessaire pour garantir que le signataire et le vérificateur appliquent l'algorithme de hachage sur exactement les même données. Un algorithme de canonicalization prend en entrée un contenu XML et fournit en sortie un tableau d'octets représentant la forme canonique ou "normalisée" de l'entrée.

Plusieurs types de détails dans un documents XML peuvent changer d'un système à un autre sans toutefois changer l'interprétation du document. Citons par exemple :

– Des détails liés aux propriétés de bases de XML. Par exemple, la présence des attributs déclarés avec des valeurs par défaut est optionnelle. La normalisation consiste à les ajouter systématiquement avec leurs valeurs par défaut. Un autre exemple : les espaces en dehors de la racine du documents sont supprimés car supposés insignifiantes.

– Des détails liés à l'application utilisée pour la lecture et l'écriture du document XML. Les deux modèles les plus utilisés sont DOM[5] et SAX[24] (voir la section 6.1). Dans les deux cas, plusieurs détails du document XML peuvent être perdu entre la lecture et l'écriture comme l'ordre des attributs, les espaces insignifiants (entre les balises, entre les attributs...), etc.

Avec la canonicalization, si un document XML est changé par une application sans modifier sa forme canonique, le document changé et le document initial peuvent être considérés comme "équivalents" par les applications les utilisant.

La recommandation W3C de XMLDSIG[17] recommande l'utilisation de l'algorithme dit "canonicalization inclusive"[2]. Cet algorithme est aussi définit par une recommandation W3C.

Prenons comme exemple d'entrée le document XML suivant :

L'entrée de la canonicalization

```
<doc>
   <norme   titre=''XML''    auteur=''W3C''> XML c'est la mode ! </norme>
   <info>   BlaBlaBla   TotoTiti   </info>
</doc>
```

La sortie de la canonicalization est le document suivant :

La sortie de la canonicalization

```
<doc>
   <norme auteur=''W3C'' titre=''XML''> XML c'est la mode ! </norme>
   <info>   BlaBlaBla   TotoTiti   </info>
</doc>
```

Notez principalement deux types de modifications introduites par la canonicalization :

1. Les attributs de *norme* ont été ordonnés alphabétiquement.
2. Les espaces insignifiants entre ces attributs ont été supprimés. Notez aussi que les espaces dans l'élément *info* ont été gardés car supposés signifiants.

Prenons maintenant un exemple plus compliqué faisant intervenir les espaces de noms (voir section 3.2.1) :

L'entrée de la canonicalization

```
<n0:default xmlns:n0="http://www.A.com">
      <n1:elem1 xmlns:n1="http://www.B.com">
          content
      </n1:elem1>
</n0:default>
```

Le résultat de la canonicalization de l'élément *n1* :

La sortie de la canonicalization de *n1*

```
<n1:elem1 xmlns:n0="http://www.A.com"
          xmlns:n1="http://www.B.com">
       content
</n1:elem1>
```

Notez que maintenant *n1* a acquit une nouvelle déclaration de nom d'espace : $n0 = "http : //www.A.com$ qui initialement était déclaré dans son père *n0*. Ce comportement est à l'origine de la qualification *Inclusive* de cet algorithme de canonicalization. En effet, il “inclue” les déclarations des nom d'espaces de tous les ancêtres d'un élément à la forme canonique de celui ci. Cela permet de s'assurer que les descendants de l'élément sont toujours associés à leur espace de noms initial même si celui-ci est déclaré dans un parent qui n'est pas dans le résultat de la canonicalization.

Ce comportement peut poser des problèmes pour certains applications de la signature XML. En particulier lorsque on souhaite changer l'enveloppe du document signé ou de la signature elle même tout en gardant sa validité. Par exemple, si on veut transporter l'élément *n1* de l'exemple précédent dans une enveloppe SOAP[25] :

Transporter *n1* dans une enveloppe SOAP

```
 <SOAP:Envelope xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/">
 <n1:elem1 xmlns:n1="http://www.B.com">
         content
 </n1:elem1>
</SOAP:Envelope>
```

Le nouveau résultat de la canonicalization est :

Résultat de la canonicalization de *n1* dans l'enveloppe SOAP

```
<n1:elem1 xmlns:n1="http://www.B.com"
          xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/">
       content
</n1:elem1>
```

On voit que la déclaration de l'espace des nom de *n0* ($http : //www.A.com$) a été remplacé par la déclaration de l'espace de nom de l'enveloppe SOAP, celui-ci étant le nouveau père de l'élément *n1*. Ce changement invalide évidemment la signature si elle a été calculé en utilisant la première forme canonique de *n1*.

Pour remédier à ce type de problème, un nouveau algorithme de canonicalization dit “Canonicalization exclusive”[7] a été définit par une autre recommandation W3C. Comme son nom l'indique cet algorithme évite d'ajouter systématiquement les déclarations des noms d'espaces des ancêtres de l'élément à canonicalizer. Seulement, les noms d'espaces qui sont effectivement utilisés par l'élément et ces descendants sont gardés. Ce type d'algorithme résout le problème de changement d'enveloppe évoqué ci-dessus. la forme canonique de l'élément *n1* est maintenant indépendante de son enveloppe et est dans tous les cas :

Résultat de la canonicalization Exclusive de *n1*

```
<n1:elem1 xmlns:n1="http://www.B.com"
       content
</n1:elem1>
```

Appliquer l'algorithme de signature

L'entrée de l'algorithme de signature est la clé privé du signataire et le haché de la forme canonique de l´élément *SignedInfo*. Le résultat de l'algorithme forme l'élément *SignatureValue*. Ces deux éléments (*SignedInfo* et *SignatureValue*) sont les seuls obligatoires dans toute signature XML. Cependant, un signataire peut vouloir fournir des éléments cryptographiques (certificats, CRL...) pour permettre au destinataire de son message de vérifier la signature et son authenticité. Ces information peuvent être transportés dans l'élément *KeyInfo*. A noter qu'une signature XML garantit simplement qu'un contenu donnée a été signé par une clé privé donnée (donc garantit l'intégrité du message et son authentification) mais ne garantit pas l'identité du signataire. Cependant, elle permet de transporter les information nécessaires pour permettre au destinataire de le vérifier par des moyens extérieurs (par vérification d'une chaîne de certificats menant du certificat du signataire à une CA de confiance par exemple).

Résumons finalement les étapes de production d'une signature XML :

1. Assembler l'élément *SignedInfo* :
   (a) Pour toute ressource à signer, former un élément *Reference* avec l´URI, les algorithmes de transformations, l'algorithme de hachage et la valeur du haché.
   (b) Ajouter les éléments *SignatureMethod* et *CanonicalizationMethod* spécifiant respectivement l´algorithme de signature et l´algorithme de canonicalization.
2. Normaliser la représentation XML du *SignedInfo* en appliquant l'algorithme de canonicalization déjà choisit.
3. Appliquer l'algorithme de signature au résultat de la canonicalization pour créer l'élément *SignatureValue* avec la valeur de la signature.
4. Éventuellement ajouter des information de validation dans un élément *KeyInfo*.

4.2.2 Types des signatures XML

XMLDSIG permet de signer tout type de document numérique. Le contenu signé et la signature peuvent être soit détachés soit attachés au même document. Le dernier cas est même possible quand la ressource signée est au format binaire en utilisant une transformation qui permet de l'encoder en une représentation textuelle compatible avec le document XML contenant la signature. L'encodage en Base64[22] est un exemple de telles transformations.
Selon la position relative de la signature et le contenu signé, XMLDSIG permet de créer trois types de signatures adaptés à des contextes d'utilisation différents.

Signature enveloppée :

Une signature XML est dite "Signature enveloppée" si la racine de la signature est un descendant de la racine du contenu XML signé (voir figure 4.1). Ce type de signature est utile lorsque la signature est transporté par un protocole lui même basé sur XML (citons par exemple SOAP[25]*.

Le contenu signé enveloppe tout le corps de la signature XML qui va être donc compris dans le calcul du haché. Cependant, le corps de la signature est censé être modifié après le calcule de la signature (c'est à dire après le calcul de tous les hachés) pour insérer l'élément *SignatureValue*. Ce qui impliquerait évidemment l'invalidation de la signature. Pour remédier à ce problème la recommandation XMLDSIG a prévu une transformation spécifique pour les signatures enveloppée.

Le comportement de cette transformation est simple : Elle enlève de son entrée l'élément *Signature* (avec tous ses descendants) contenant la transformation.

*SOAP : Simple Object Access Protocol

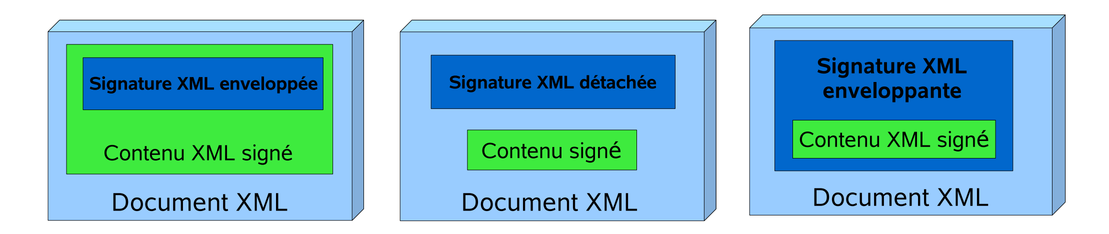


Fig. 4.1 – Les types de signatures XML.

Signature enveloppante :

Une signature XML est dite “signature enveloppante” si la racine du contenu signé est un descendant de la racine de la signature (voir la figure 4.1). XMLDSIG définit un conteneur spécifique *object* pour ce type de signature (voir 4.3).

Signature détachée :

Une signature XML est dite “signature détachée” si elle n’est ni enveloppé ni enveloppante (voir la figure 4.1). Il y a deux types de signature détachées :
- La ressource signée et le document contenant la signature XML sont totalement détachés (ressource distante).
- La ressource signée et la signature sont sur le même document XML mais dans deux niveau d’encapsulation différents (ni la racine de la signature n’est un ancêtre du contenu signé, ni la racine du contenu signé n´est un ancêtre de la signature).

4.2.3 Validation des signatures XML

La validation d’une signature XML peut se faire en deux étapes. La première consiste à valider les hachés des ressources signés définis dans les éléments *Reference*. La deuxième étape consiste à valider la valeur de la signature contenue dans l´élément *SignatureValue*.

Étape 1 : Validation des éléments *Reference*

Cette étape consiste à vérifier l’intégrité de toutes les ressources signées en validant leurs empreintes numériques. Pour chaque élément *Reference* :

1. On récupère le contenu signé en résolvant l’URI.
2. Si des éléments *Transform* sont présents on applique en séquence les algorithmes de transformation spécifiés. Quand une transformation requiert en entrée un tableau d´octets alors que la sortie de la transformation précédente est un contenu XML, on essaie de le convertir en utilisant l’algorithme de canonicalization INCLUSIVE qui donne toujours en sortie un tableau d´octets.
3. On applique l’algorithme de hachage spécifié dans l´élément *DigestMethod* à la sortie de la séquence des transformations.
4. On compare la valeur obtenue avec celle contenue dans l´élément *DigestValue*. Si elle sont égales, la *Reference* est valide, sinon on peut conclure que le contenu signé a été altéré et donc la signature n´est pas valide.

Étape 2 : Validation de la valeur de la signature

Cette étape consiste à vérifier la valeur de la signature contenue dans l´élément *SignatureValue*. L'entrée à ce processus est la représentation XML de l´élément *SignedInfo* et les information cryptographiques contenus dans l´élément *KeyInfo* s´il est présent. Il peut s'écrire en trois étapes :

1. Récupérer la clé publique du signataire : l'application peut la récupérer directement du *KeyInfo* si elle y est présente ou part d'autres moyens extérieurs à la signature. L´élément *KeyInfo* peut aussi transporter des donnée de validations (chaînes de certificats, identifiant, CRL...) pour permettre la vérification de la validité de la clé publique et de l'identité associée.
2. Calculer la forme canonique de l´élément *SignedInfo* en utilisant l´algorithme de canonicalization spécifié dans l´élément *CanonicalizationMethod*.
3. Enfin, appliquer l'algorithme de la signature sur la forme canonique obtenue en utilisant la clé publique. La signature ne peut être supposée valide que si la valeur obtenue est égale à la valeur transportée dans l´élément *SignatureValue*.

Autres considérations de validation :

Les deux étapes précédentes ont abordé le problème de la validation de la signature par un point de vue purement cryptographique. Cependant, une signature XML peut être considéré invalide pour plusieurs d'autres raisons qui peuvent ou non impliquer l´échec d´au moins une des étapes précédentes. Citons par exemple :

– **Validation du document de la signature par rapport à un schémas XML :** Cette validation consiste à s'assurer que la structure XML de la signature est conforme au schémas établie par la recommandation W3C des signatures XML. Une signature peut donc être rejeté si elle contient des structures non conformes à ce schémas même si les deux étapes précédentes ont été validées.
– **Impossible de résoudre une URI** : les ressources signées sont adressées avec des URI. L´échec de la résolution d'une de ces URI entraîne naturellement l´échec de la première étape. D´autre part, l'application peut bien ignorer cet attribut si elle est capable de récupérer directement (à partir d´un cache par exemple) le contenu signé. XMLDSIG permet même dans ce cas d´omettre l´attribut URI. Cependant, pour éviter toute ambiguïté, au plus un élément *Refrence* sans URI peut être présent dans une signature XML.

4.3 Vue structurelle : syntaxe de XMLDSIG

La recommandation W3C spécifie de façon exhaustive tous les éléments d´une signature XML. Les structures sont définis grâces à des DTD[34] (Document Type Definition) et des schémas XML (voir section 3.2.1). Il est cependant conseillé d´utiliser les schémas car contrairement aux DTD ils supportent la notion des espace de noms. Le schéma complet des signatures XML est donné dans l´annexe 7.3.2.

La recommandation définit également l'espace de noms des signature XML. Une signature contenant un élément extérieur à cet espace de noms peut être considérée invalide. Cependant, XMLDSIG prévoit un élément spécifique pour contenir toute structure extérieur à l´espace de noms de la signature. Ce conteneur va être détaillé dans les sections suivantes.

La figure 4.2 donne la structure générale d'une signature XML (où '+' indique que l'élément peut apparaître une ou plusieurs fois, '?' indique que l'élément peut apparaître au plus une fois et '*' indique que l'élément peut apparaître zéro ou plusieurs fois).

```
<Signature ID?>
    <SignedInfo>
      <CanonicalizationMethod/>
      <SignatureMethod/>
      (<Reference URI? >
        (<Transforms>)?
        <DigestMethod>
        <DigestValue>
      </Reference>)+
    </SignedInfo>
    <SignatureValue>
   (<KeyInfo>)?
   (<Object ID?>)*
  </Signature>
```

Fig. 4.2 – La structure simplifiée d'une signature XML

On va se limiter dans cette section aux éléments les plus importants. l´étude exhaustive des éléments XMLDSIG peut nécessiter tout un rapport. On abordera aussi certains éléments qui, bien qu'ils sont optionnels, peuvent avoir plusieurs applications pratiques.

L´élément *SignedInfo*

L'élément *SignedInfo* contient la structure XML effectivement couvert par la signature. Il est constitué par 3 type d´éléments :

L´élément *Reference*

Un *SignedInfo* doit contenir au moins un élément *Reference* désignant la ressource à signer par un attribut URI. Cet élément permet aussi de qualifier la ressource par d'autres attributs qui peuvent aider l'application à résoudre et décoder l'URI. A savoir :

- Encoding : décrit l´encodage de la ressource.
- Type : indique le type MIME de la ressource (par exemple image/png).

Comme ça été abordé dans les sections précédentes, un élément *Reference* peut contenir une séquence d'élément *Transform* désignant chacun un algorithme de transformation à appliquer au contenu signé.

L´élément *Transform*

Cet élément a une structure très simple. Il contient seulement un attribut *Algorithm* identifiant l'algorithme de transformation à utiliser. Il peut aussi contenir d'autres élément qui dépendent de la transformation désignée. Tous les algorithmes en XMLDSIG sont définis avec des URIs. La recommandation du W3C identifie un certains nombre d'algorithmes mais des algorithmes extérieures peuvent bien être utilisés. Une recommandation W3C ultérieure[21] définis un ensemble d'URIs pour des algorithmes de transformation, hachage et signature qui peuvent être utilisé dans XMLDSIG.

Citons comme exemple les algorithmes de transformation suivant :

Les algorithmes de canonicalization : En effet tout algorithme de canonicalization peut être utilisé comme un algorithme de transformation (voir la section 4.2.1).

Transformation pour les signatures enveloppées : Cette transformation a été définit spécifiquement pour les signatures enveloppées (voir la section 4.2.2).

Transformation Base64 : Cette transformation permet de coder un contenu binaire en un ensemble de caractères transportable dans une signature XML et vice-versa. Elle est utilisée pour signer un contenu binaire tout en l'incluant dans le document XML de la signature. Les règles de l'encodage et décodage à utiliser sont définit dans la RFC 2045[13].

Transformation XPath : Cette transformation permet d'utiliser des expressions XPath (voir section 3.2.1 pour sélectionner certains parties dans le document désigné par l'URI. Cette transformation est un outil très puissant pour les applications qui signent par exemple des document HTML. En utilisant, des expressions XPath, l'utilisateur peut par exemple signer uniquement certains champ textes dans une page HTML.

Les éléments *SignatureMethod* et CanonicalizationMethod

Ces éléments désignent respectivement les algorithmes de signature et canonicalization à utiliser. Comme pour les transformations, ces algorithmes sont définis avec des URIs.

Pour les algorithmes de signature, XMLDSIG définit deux URI mais d'autres peuvent bien être utilisés :

- "$http://www.w3.org/2000/09/xmldsig\#dsa-sha1$" : désigne un hachage en SHA1 suivi par un chiffrement DSA.
- "$http://www.w3.org/2000/09/xmldsig\#rsa-sha1$" : désigne un hachage SHA1 suivi par un chiffrement RSA.

L´élément *KeyInfo*

Cet élément est optionnel et offre plusieurs moyens pour intégrer dans la signature des données cryptographiques de validation. Le signataire peut par exemple donner explicitement sa clé publique ou un certificat le contenant ou même désigner une ressource distante où le vérificateur peut trouver cette information. Il est à la charge de l´application qui vérifie la signature de décider de degré de la confiance à donner aux différentes structures contenues dans cet élément.

On va décrire dans cette section certains mais pas toutes les structures du *KeyInfo* permettant de transporter des données de validation. Le schémas XML de cet élément prévoit aussi un élément ouvert qui peut être utiliser pour incorporer des données cryptographiques spécifiques à l'utilisateur non définis dans XMLDSIG.

L´élément *KeyName*

Cet élément peut simplement contenir un identifiant qui identifie de façon directe ou indirecte la clé publique du signataire. Cet identifiant peut être : un index de clé, un nom distinctif (DN[†]), une adresse émail, etc.

Les éléments *RSAKeyValue* et *DSAKeyValue*

Ces deux éléments permettent respectivement de transporter des clés publiques RSA et DSA. Au plus un de ces deux éléments peut être présent et il doit correspondre au algorithme de signature utilisé. Les paramètres de la clé sont naturellement encodé en Base64.

[†]DN : Distinguished Name.

L´élément *X509Data*

Cet élément peut transporter plusieurs structures reliées aux standards X.509[15]. Parmi ces structures on cite :

Des identifiants de certificat : Un signataire peut désigner le certificat contenant sa clé publique par plusieurs façons :
- en donnant le nom distinctif de l´autorité de certification qui l´a généré et le numéro de série du certificat dans un élément *X509IssuerSerial*.
- en donnant le nom distinctif du propriétaire du certificat dans un élément *X509SubjectName*. A noter, que cette information toute seule peut ne pas être suffisante pour identifier un certificat. En effet, une personne peut avoir plusieurs certificats différents pour des utilisation différentes (et naturellement des pairs de clés différentes).
- en donnant l´extension SubjectKeyIdentifier de X509v3[15] qui fournis un moyen d'identifier un certificat contenant une clé publique particulière. Cette extension peut être transporté dans élément *X509SKI*.

Des certificats : Un signataire peut aussi fournir directement le certificat contenant sa clé publique ou une chaîne de certificats aboutissant au certificat contenant la clé. Un certificat est encodé en Base64 et transporté dans un élément *X509Certificate*.

Des CRLs : De même, une signature peut contenir une ou plusieurs CRL liées aux certificats données. Ces CRLs sont encodés en Base64 dans des éléments *X509CRL*.

L´élément *Object*

Cet élément est définit dans XMLDSIG comme un élément de contenu ouvert. Bien qu´optionnel il a plusieurs utilisations pratiques. Il peut être utilisé pour intégrer un contenu XML dont les structures sont extérieures au espace des noms de XMLDSIG. En particulier, il est utilisé par les signatures enveloppantes pour "envelopper" le contenu signé. Il également utilisé comme conteneur de plusieurs structures optionnelles. Certains de ces structures seront abordées dans les sections suivantes.

L´élément *SignatureProperties*

Cet élément optionnel permet de transporter des propriétés qualifiant les ressources signées ou le signataire. La notion de priopriété de signature peut être vue comme l´équivalent des attributs CMS (voir section ??). Il permet par exemple de porter des information sur l´heure de signature, le numéro de série du matériel cryptographique utilisé pour la génération de la signature, etc.

Cet élément doit obligatoirement être contenu dans un élément *Object*. Il peut être facultativement signé pour transporter des propriétés signées.

Chapitre 5

XAdES : XML ADvanced Electronic Signatures

XAdES[32] est une extension de XML Digital Signatures. L´extension concerne notamment le domaine de la non-répudiation, en définissant des formats XML pour les “Signatures électroniques avancées” susceptibles de rester valides pendant de grandes périodes, conformément à la “Directive Européenne 1999/93/EC” [4]. XAdES est le résultat des traveaux de l´ETSI dans ce domaine. Une note W3C reprend ces spécifications, en vue d´une future recommandation W3C.

5.1 Contexte juridique

Le cadre juridique définissant le statut de la signature électronique en France, et plus généralement en Europe, est le résultat de la la directive européenne 1999/93/CE[4]. Les principaux textes sont les suivants :

- 1999 : Directive Européenne 1999/93/CE
- 2000 : Loi n 2000-230 du 13 mars 2000 : Prise en compte de la signature électronique au sein du code civil.

La directive européenne définit les signatures numériques par : “des données sous forme électronique, qui sont liées ou associées logiquement à d’autres données électroniques et qui servent de méthode d’authentification”[4]

XAdES ajoute à XMLDSIG les information nécessaires pour être conforme à ces décrets : “Une signature électronique évoluée en accord avec le présent document (XAdES) peut [...] être utilisée pour arbitrage en cas de litige entre le signataire et le vérificateur, litige qui peut survenir plus tard, éventuellement des années après.” [32].

Une signature XAdES permet d’incorporer des nouvelles structures dans une signature XMLDSIG pour fournir des indications pouvant être traitées afin d’obtenir l’assurance qu´une signature a été produite sous un certain engagement (explicite ou implicite), sous couvert d’une politique de signature, à un instant donné, par un signataire identifiable par un nom ou un pseudonyme et éventuellement un rôle.

Syntaxiquement XAdES est définit au dessus de XMLDSIG. Toutes les structures de XAdES sont incorporées dans une signature XMLDSIG en utilisant l´élément *Object* comme conteneur. XAdES définit un grand nombre de propriétés qui peuvent qualifier le signataire ou les ressources signées et qui peuvent être signées ou non. Selon, les propriétés présentes XAdES définit six forme différentes. Dans ce chapitre on va décrire ces différents formes sans entrer dans les détails syntaxiques (l´annexe

Signature électronique évoluée (XAdES)

Infos signées | Signature | Infos clé | Propriétés signées | Propriétés non signées

FIG. 5.1 – Signature numérique évoluée.

7.3.2 donne le schémas XML complet des signatures XAdES). Ensuite, on détaillera l´évolution d´une signature XAdES d´une forme basique à une forme plus complète.

5.2 Les type de signatures XAdES

Comme on l´a déjà dit XAdES est construit comme une extension de XMLDSIG. L´élément racine de toutes les structures XAdES est l´élément *QualifyingProperties*. Cet élément doit être contenu dans l´élément *Object* de XMLDSIG. Selon les propriétés présentes, une signature est dite conforme à une certaine forme. XAdES permet de créer six formes différentes[32].

5.2.1 XAdES-BES (Basic Electronic Signature)

C´est la forme de base de XAdES. Une signature est dite BES si elle vérifie l´une des deux règles suivantes qui sont fonctionnellement équivalentes :
- La signature contient la propriété signée *SigningCertificate* : Cette propriété permet de transporter et signer l´identifiant du certificat du signataire.
- La signature contient le certificat du signataire dans son élément *KeyInfo* et ce dernier est couvert par la signature.

Chacune des règles précédentes permet de signer le certificat du signataire. Cette forme de signature permet de contrer toute attaque qui essaie simplement de substituer le certificat du signataire par un “mauvais”certificat.

Facultativement, une signature BES peut contenir une ou plusieurs des propriétés suivantes :
- **Propriétés signées qualifiant la signature :**
  - *SigningTime* : La date de la signature.
  - *SignatureProductionPlace* : Le lieu de production de la signature.
  - *SignerRole* : Le rôle du signataire. Pour certaines applications le rôle du signataire (exemple : sa fonction au sein d´une entreprise) au moment de sa signature peut être plus important que son nom.
- **Propriétés signées qualifiant les ressources signées :**
- *DatObjectFormat* : Qualifie le format du contenu signé.
- *CommitmentTypeIndication* : Indique l’engagement sous lequel le signataire a produit la signature. Cet engagement peut être indiqué de façon explicite en donnant un identifiant d´engagement ou implicite en le déduisant à partir de la sémantique du contenu signé.

- *AllDataObjectsTimeStamp* : Contient la valeur de la datation de toutes les ressources signés. Le but de cette propriété est de prouver l´existence de ces ressources à la date de la signature.
- *IndividualDataObjectsTimeStamp* : Elle a le même rôle que la propriété précédente sauf qu´elle permet de dater un sous ensemble des ressources signées.
- **Propriétés non signées qualifiant la signature :**
- *CounterSignature* : Permet d´encapsuler une contre-signature. Cette propriété est équivalente à l´attribut *CounterSignature* de CMS (voir la section ??). Une contre-signature peut être une signature XMLDSIG ou XAdES ce qui permet de créer une chaîne aléatoirement longue de contre-signatures.

Une signature BES permet de transporter les information minimales pour être conforme à la directive européenne des signature numériques. Cependant, elle ne contient pas les information de validation nécessaire pour garantir sa validité à long terme. Par exemple, une signature BES peut ne plus être vérifiable quand les certificats et les CRL nécessaires à sa validation ne sont plus disponibles.

5.2.2 XAdES-EPES (Explicit Policy Electronic Signature)

Cette forme peut être construite au dessus de XMLDSIG ou XAdES-BES. Elle ajoute la propriété *SignaturePolicyIdentifier*. Cette propriété est signée et elle permet de transporter l´identifiant de la politique de validation. Une politiques de validation de signature spécifie des règles et des contraintes à suivre au cours du processus de validation. Elle peut spécifier par exemple les propriétés obligatoires que le signataire doit fournir, les algorithmes à utiliser, etc.

5.2.3 XAdES-T (Electronic Signature with Time)

les formes précédentes sont conformes aux obligations légales définies par la directive européenne mais ne garantissent pas la non-répudiation jugée importante par cette directive[4]. Cette forme ajoute une date à XAdES permettant de prouver dans un litige ultérieur l'existence de la signature à une date donnée. Cette date peut être créer par le signataire au moment de la production de la signature, ou par le vérificateur dès la réception de la signature. Syntaxiquement, cette forme ajoute la propriété non signée *SignatureTimeStamp* à une signature XAdES-BES.

Cette forme permet donc de garantir une authentification et une protection de l'intégrité minimales ainsi qu'une protection contre la répudiation.

5.2.4 XAdES-C (Electronic Signature with Complete Validation Data)

Cette forme permet de transporter les données de validation nécessaires à la vérification de la signature. Elle définit des conteneurs pour transporter des identifiants de certificats, CRLs et/ou réponses OCSP* (voir la section 3.1.2). A noter que cette forme ne transporte que des références et non pas les données réelles qui sont naturellement beaucoup plus volumineuses. Cette forme peut être ajouté par le signataire ou le vérificateur dès que l´ensemble des données de validation devient disponible.La figure 5.2 illustre une signature XAdES-C. Syntaxiquement, cette forme ajoute les deux propriétés non signées suivantes : *CompleteCertificateRefs* et *CompleteRevocationRefs*.

*OCSP : Online Certificate Status Protocol.

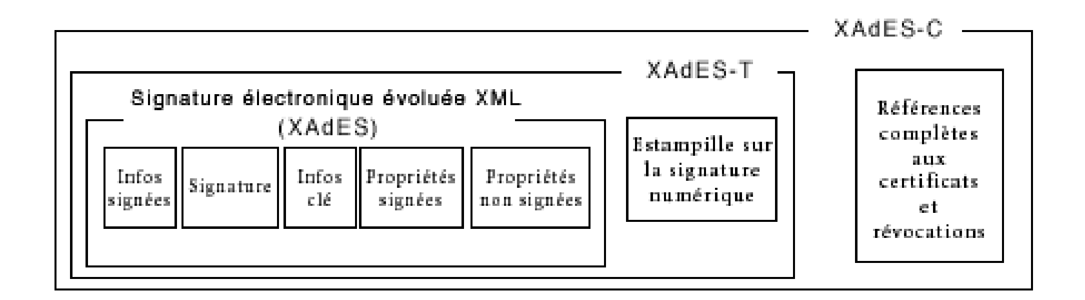


Fig. 5.2 – Signature numérique évoluée : forme XAdES-C.

5.2.5 XAdES-X (Electronic Signature with with eXtended validation data)

Cette forme peut être créé par le signataire ou le vérificateur en datant l´ensemble des données de validation transportées par la forme précédente. Elle permet de garder la validité de la signature même si une partie de ces données sont compromis après la production de la signature ou après la première vérification. Pour une protection plus maximale, un vérificateur peut même incoprpoer la totalité des données de validation dans la signature ce qui construit la forme suivante.

5.2.6 XAdES-X-L (Electronic Signature with eXtended validation data for the long term)

Cette forme permet de transporter la totalité des information de validation (voir la section 4.3). Elle est nécessaire pour garantir la validité de la signature à long terme et en particulier dans le cas où les données de validation ne sont pas stockés ailleurs.

5.2.7 XAdES-A (Electronic Signature with archiving validation data)

Pour une protection maximale à long terme, cette forme permet de dater l´ensemble des données de validation transportées par la forme précédente. Cette datation peut protéger la signature contre l´affaiblissement au long terme des différents données cryptographiques (algorithmes, clés, etc) utilisées pour valider la signature. Syntaxiquement, elle ajoute la propriété non signée *ArchiveTimeStamp* à une signature XAdES-X-L. Plusieurs occurrences de cette propriété peuvent apparaître dans une même signature. Un vérificateur peut effectuer une nouvelle datation dès que la datation ultérieure s´affaiblit.

Chapitre 6

Implémentation

Ce chapitre va présenter les différents développements effectués au cours de stage. Tous les bibliothèques ont été écrite en Java.

La taille de ce rapport étant limité, on va seulement détailler les principaux bibliothèques, à savoir : XML, XMLDSIG et XAdES.

Tout le code source écrit au cours de ce stage étant considéré confidentiel par l'entreprise, ce chapitre ne va pas en présenter. Je mets cependant en annexe un mini-manuel d'utilisation rédigé (en anglais) sur le wiki de l'entreprise pour XMLDSIG. Il y a également en annexe une réflexion sur l'architecture générale d'un serveur de signature utilisant DSS rédigé (en anglais) sur le wiki de l'entreprise.

6.1 XML

Étant déjà largement utilisé sur Internet, une grande multitude d'implémentations libres ou commerciales existe déjà pour XML. Ces différentes implémentations sont conformes à l'un des deux modèles les plus connus de traitement de contenu XML, à savoir : DOM (Document Object Model)[5] et SAX (Simple API for XML)[24]. La différence majeure consiste dans la manière le document XML d'entrée est traité :

- SAX permet de faire des traitements basés sur des évènements[24](comme lecture d'une balise ouverte, lecture d'un attribut...). Il n'est pas nécessaire de lire la totalité du document pour effectuer un traitement.
- DOM offre une vue logique globale sur le document XML en le représentant sous forme d'un arbre[5]. Pour cela la totalité du document est mise en mémoire.

Pour la validation de la signature, on a évidemment besoin de la totalité du document (pour l'évaluation des expressions XPath, pour la résolution des URI locales, etc.). D'où le choix du modèle DOM.

6.1.1 Solutions existantes

Les implémentations DOM sont assez abondantes. Les plus connus sont Xercer*, JDOM† et XOM‡. Il n'était pas dans le but du stage de développer une nouvelle implémentation. Java lui même offre à partir de sa version 1.5 plusieurs outils XML y compris XPath. Cependant, pour des

*Apache Xerces : http ://xerces.apache.org/

†JDOM : http ://www.jdom.org/

‡XOM : http ://www.xom.nu

raisons de compatibilités avec ses propres applications, l'entreprise souhaite une compatibilité avec au moins Java 1.2. La bibliothèque choisie est XOM[39].

XOM est une implémentation récente du modèle DOM[5]. Il fonctionne à partir de Java 1.2 et il offre tout le nécessaire pour XMLDSIG :
- le support de XPath.
- La gestion des espace de noms.
- la canonicalisation.
- La validation avec des schémas XML.

6.1.2 Solution implémentée

La bibliothèque développée utilise donc XOM pour offrir des fonctionnalités de plus haut niveau. Elle implémente les classes de bases qui vont servir pour les bibliothèques XMLDSIG et XAdES.

Diagramme des classes

Le diagramme des classes est donné dans la figure 6.1. Cette bibliothèque est composé de 4 packages :

1. **com.cryptolog.xml** : Les classes de base. Les plus importantes :
   - *XMLData* : c'est une classe abstraite qui représente un contenu XML sous forme d'un ensemble d'éléments XML ou une suite d'octets (représentant l'ensemble d'éléments du contenu XML).
   - *NodeSetData* : hérite de la classe précédente pour représenter un ensemble d'éléments et/ou documents XML. Elle offre des méthodes pour le traitement de ces données (exécution de requêtes XPath, sélection de sous ensemble d'éléments...).
   - *OctetStreamData* : hérite de la classe abstraite *XMLData* pour représenter le contenu XML sous forme d'un flux d'octets.
   - *XMLStructure* : représente une structure XML de base. Elle sera héritée par toutes les classes représentant des éléments XML.
2. **com.cryptolog.xml.prop** : contient les classes de base pour la gestion des propriétés en XML. Principalement :
   - *XMLProperty* : représente une propriété XML. Une propriété est un couple nom/valeur. Elle peut être désignée de façon unique par un OID ou par un couple nom de tag/espace de noms.
   - *XMLPropertyReader* : définit un lecteur de propriétés XML pour le décodage de ces propriétés. L'intérêt majeur de cette classe est que la liste des propriétés reconnues peut être alimentée de façon dynamique en appelant *addPropertyClass()* avec le nom de la balise de début de la propriété et la classe à instancier si cette balise a été trouvé dans le document XML.
3. **com.cryptolog.xml.transforms** : contient des implémentations de plusieurs transformations basés sur un contenu XML. On cite par exemple : les algorithmes de canonicalisation (*ExclC14nAlgorithm* et *C14nAlgorithm*), la transformation XPath[40](*XPathTransform*), la transformation Base64[22](*Base64Transform*). Toutes ces classes héritent de la classe abstraite *TransformAlgorithm* qui implémente les méthodes de base d'une transformation. Chaque classe de transformation particulière doit implémenter une méthode *transform()* qui prend en entrée une instance de *XMLData* et donne en sortie le résultat de la transformation aussi dans un *XMLData*. Chaque transformation spécifie les types des données d'entrée et de sortie (NodeSetData ou OctetStreamData) ainsi que les différents paramètres en instanciant à la

construction une implémentation particulière de la classe *TransformParameterSpec*.
Ce package implémente aussi une classe *TransformFactory* qui permet d'obtenir un algorithme de transformation à partir de son URI et éventuellement ses paramètres. En particulier, les algorithmes qui n'ont aucun paramètre (comme la transformation Base64 et la plupart des algorithmes de canonicalisation) sont utilisés comme des singletons[§].

4. **com.cryptolog.xml.types** : contient des classes représentant des structures XML de base qui seront utilisés dans les autres bibliothèques. On cite par exemple :
   – *XMLDateTime* : qui implémente la représentation XML normalisée[37] de la date.
   – *XMLEncapsultedPKIData* : représente une structure XML qui transporte un contenu binaire encodé en Base64[22].

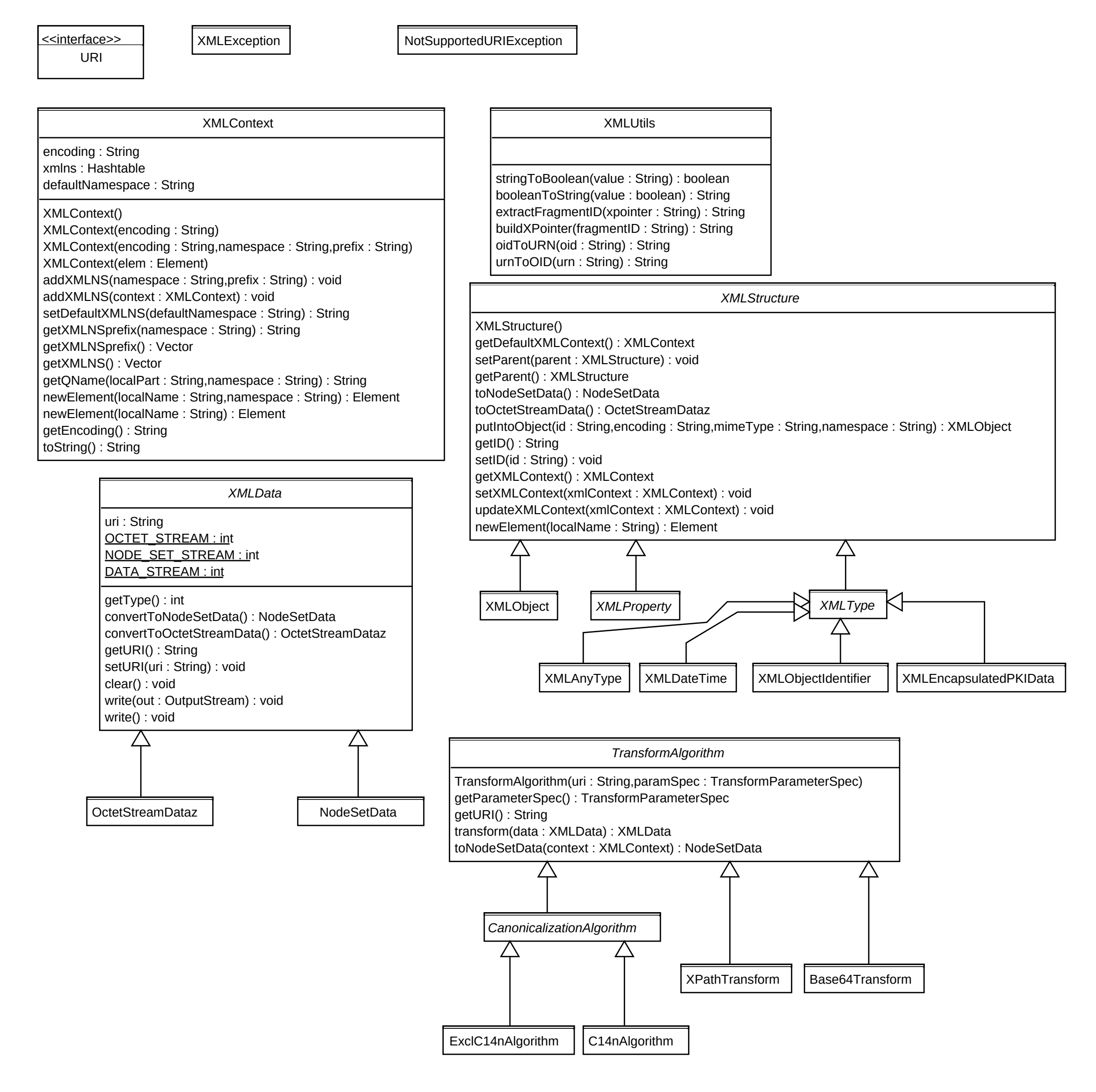


Fig. 6.1 – Diagramme de classe du package xml.

[§]Un singleton est un design pattern très connu et est utilisé pour une classe dont l'état intérieur (les valeurs des attributs) ne peut changer et est toujours le même dans tous les cas d'utilisation.

6.2 XMLDSIG

6.2.1 Solutions existantes

Il existe actuellement plusieurs implémentations de XMLDSIG plus au moins complètes. Parmi ces implémentations on cite celle de Apache XML Security¶ et et de l'IAIK‖ (voir chapitre 7).

La plupart de ces implémentations sont basés sur la JSR (Java Specification Request) 105 de Java**. Celle ci est intégrée dans Java à partir de sa version standard 1.6. L'entreprise a entrepris le développement de sa propre implémentation pour divers raisons et plus particulièrement :

- avoir un support d'au moins java 1.2.
- utiliser ses propres librairies en tout ce qui concerne la cryptographie (algorithmes, certificats,...).

6.2.2 Solution implémentée : fonctionnalités

L'API implémentée supporte la plupart des éléments de base définis dans la recommandation W3C (voir chapitre 4). Elle permet plusieurs opérations sur les signatures XML. Principalement, l'API permet de :

1. **produire** une signature XML en fournissant le document à signer, seulement son haché avec l'algorithme de hachage utilisé ou encore un format transformé du contenu original avec les algorithmes de transformation appliqués. L'API permet également la production des trois types de signatures XML (à savoir détachée, enveloppée ou enveloppante).
2. **valider** un document de signatures XML. Un document XML peut contenir plusieurs signatures ou contre-signature. L'API permet de valider une signature XML selon les règles de validation de la recommandation W3C (voir section 4.2.3). Le vérificateur peut soit fournir le document original signé soit seulement son haché.
3. **modifier** un document de signatures XML soit en ajoutant une contre-signature (en signant la valeur de la signature présente dans le document) ou une co-signature (en signant le même contenu signé par la signature présente dans le document).

6.2.3 Solution implémentée : architecture générale

Le diagramme de classe est donnée dans la figure 6.2.

On va décrire l'API en détaillant les principales opérations qu'elle permet de réaliser sur une signature XML. Un mini-manuel d'utilisation a été rédigé en anglais sur Confluence pour l'entreprise (voir annexe ??).

Production :

Il s'agit de construire un document XML de signature selon les règles syntaxiques de la recommandation W3C. Pour chaque élément définit dans le schémas (voir annexe 7.3.2) de XMLDSIG correspond une classe Java. Chacune de ces classes est dotée d'un constructeur qui prend en paramètre une structure XML pour récupérer les différents information transportés dans la structure et une méthode *toNodeSetData()* qui retourne la représentation XML de l'objet. Toutes ces classes héritent de la classe *DSigXMLStructure* qui elle hérite de la classe *XMLStructure* (du package *com.cryptolog.xml*) représentant une structure XML.

La production d'une signature XML peut se faire en trois étapes :

¶XML Apache : www.apache.org

‖IAIK : Institute for Applied Information Processing and Communication, Graz University of Technology

**JSR 105 : http ://java.sun.com/webservices/docs/1.5/xmldsig/index.html

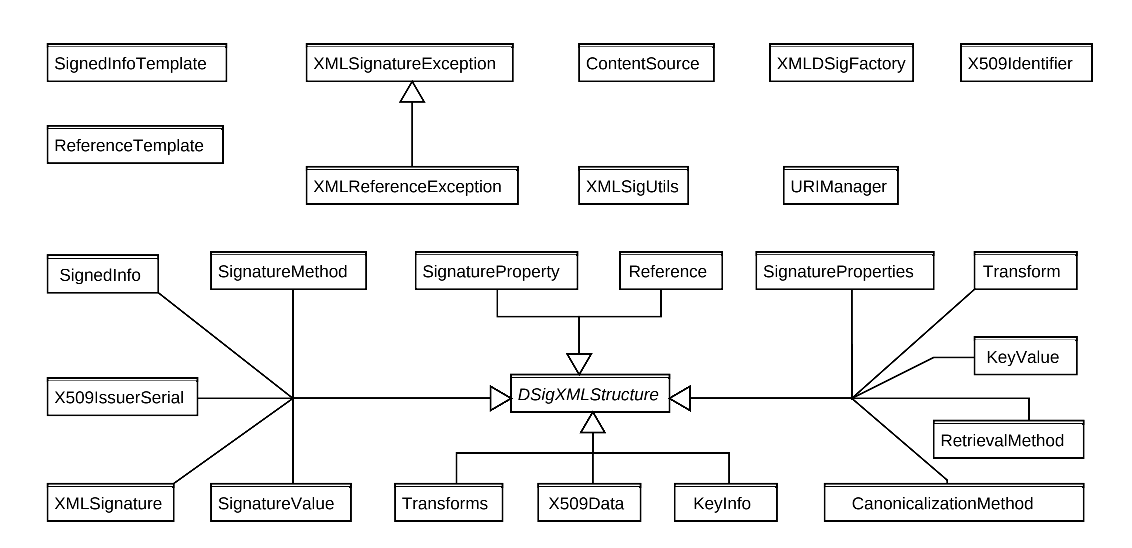


FIG. 6.2 – Diagramme de classe du package xmldsig.

1. **Initialisation du *SignedInfo* :**
   La classe de base à instancier par le signataire est *SignedInfo*. Cette classe est instanciée en fournissant au moins un algorithme de canonicalisation, un algorithme de signature et un vecteur d'objets *Reference*.

   Un objet *Reference* peut être construit avec l'URI qui désigne le contenu à signer, l'algorithme de hachage à utiliser et éventuellement une séquence d'algorithmes de transformation à appliquer sur le contenu avant le calcul de son haché. Ensuite,pour chaque référence il faut attacher le contenu à signer avec la méthode *attachContent()* qui prend en paramètre un objet *ContentSource*. Cet objet peut représenter le contenu à signer de trois façons différents :
   (a) soit explicitement la suite d'octets représentant ce contenu.
   (b) soit seulement la valeur du haché de ce contenu ainsi que l'algorithme de hachage qui a été utilisé pour la calculer.
   (c) soit une forme transformée du contenu avec la séquence des algorithmes de transformation utilisés.

   En plus du contenu, l'objet *ContentSource* définit également le type de signature voulu (enveloppante, enveloppée ou détachée).

   Le signataire peut ensuite réaliser les opérations suivantes sur l'objet *SignedInfo* :
   – Ajouter un nouveau objet représentant un élément *Reference* pour signer plusieurs documents dans une seul document de signature.
   – Ajouter une propriété signée en ajoutant un objet *SignatureProperty* représentant l'élément *SignatureProperty*.
   – Obtenir la forme canonique de l'élément *SignedInfo*.

2. **Production de la valeur de la signature :**
   Une fois l'objet *SignedInfo* est bien paramètré (avec toutes les ressources et les propriétés à signer), le signataire peut appeler la méthode *sign()* pour obtenir l'objet *XMLSignature* représentant une signature XML. La méthode *sign()* prend en paramètre :
   – Un objet *KeyInfo* représentant l'élément *KeyInfo*. Cet objet doit contenir toute les informations de validation (certificats, CRLs...) que le signataire souhaite attacher à sa signature.

- Un objet *SignatureGenerator* : Cette classe fait partie des librairies cryptographiques de l'entreprise. Elle encapsule la clé privé du signataire qui va être utiliser pour la production de la valeur de la signature.

3. **Production du document XML de la signature :**
   Après la signature, l'objet *XMLSignature* obtenu encapsule toutes les données de la signature mais aucune vrai structure XML. Notons cependant, qu'à ce stade la valeur de la signature a été déjà calculé. Donc, l'objet *XMLSignature* ne permet aucune modification sur les éléments couverts par la signature, à savoir : le *SignedInfo* et toutes les propriétés signées. En effet, la représentation XML de ces parties a été calculé à l'appel à la méthode *sign()* et ne sera pas affectée si des modifications sont faites sur les objets associés.

   Avant la génération du document XML, le signataire peut ajouter des propriétés non signées en appelant la méthode *addUnsignedProperty()* qui prend en paramètre un objet *SignatureProperty* et il peut également ajouter des contenus XML aléatoires (contenant par exemple des information supplémentaires sur la signature) en appelant la méthode générique *attach()* avec en paramètre un objet *XMLObject* qui représente un contenu XML aléatoire.

   Enfin, quand toutes les données supplémentaires ont été attachées, le signataire peut produire le document XML final en appelant la méthode *marshal()* qui écrit la représentation XML de la signature et tout les objets attachés dans un fichier ou autre support de sortie.

Les étapes précédentes permettent de paramétrer toutes les étapes de production d'une signature XML. Pour les cas les plus fréquents, la classe *XMLUtils* offre des méthodes pour construire rapidement des signatures avec les paramètres les plus courants en utilisant des templates prédéfinis de l'élément SignedInfo (voir l'annexe **??**). Un template prédéfini représente l'élément *SignedInfo* avec des valeurs par défaut (SHA1 pour le hachage, RSA-SHA1 comme algorithme de signature, la canonicalisation inclusive pour les contenus XML...).

Validation :

Il s'agit de valider les signatures présentes dans un document XML. La première étape consiste à construire des objets *XMLSignature* représentant les signatures présentes dans le document à vérifier. La classe *XMLDSigFactory* offre une multitude de méthodes *unmarshal()* qui permettent de parcourir un document XML donnée et construire des objets *XMLSignature* à partir des structures XML qu'il contient. L'objet *XMLSignature* permet ensuite d'accéder aux différentes information transportées dans la signature. La vérification pour chaque *XMLSignature* peut se faire en trois étapes :

1. Récupérer les propriétés signées et non signées ainsi que les différents objets attachés à la signature. Le traitement à faire de ces données dépend du contexte de l'application.

2. Récupérer l'objet *KeyInfo* contenant toutes les données de validation (clé publique, certificats, CRLs...). Le traitement de ces données se fait à l'extérieur de XMLDSIG. Le vérificateur doit à l'issue de cette phase sélectionner la clé publique du signataire. Autrement, la signature ne peut pas être vérifié et peut donc être supposée invalide.

3. Vérifier la valeur de la signature. Cela se fait en appelant la méthode *verify()* qui prend en paramètre la clé publique du signataire.

La classe *XMLSignature* permet également d'autres scénarios de vérifications. On peut par exemple se limiter à la vérification de la validité des éléments *Reference* en utilisant la méthode

verifyReference(). Pour cela, il faut attacher à chaque objet *Reference*, un objet *ContentSource* désignant la ressource signée. Comme pour la production, ce *ContentSource* peut contenir soit effectivement la suite d'octets représentant le contenu, soit seulement son haché ou seulement une forme transformée du contenu initial.

Contre-signature :

Le vérificateur peut également appliquer une contre-signature à une signature existante. Bien que XMLDSIG[17] ne définit pas un conteneur spécifique pour la contre-signature (cela est rattrapé par XAdES[32]), celle-ci peut être transportée dans un élément *Object* attaché à la signature. La classe *XMLSignature* fournit la méthode *countersign()* pour produire la contre-signature. Cette méthode retourne un objet *SignedInfo* spécifique contenant un seul élément *Reference* dont l'URI référence l'élément *SignatureValue* de la signature à contre-signer. En suivant les étapes de la construction de la signature abordées ci-dessus, le nouveau signataire peut produire de ce *SignedInfo* un nouveau objet *XMLSignature* qui peut être attaché à la première signature avec l'appel à la méthode *attach()*.

Co-signature :

Le vérificateur peut aussi ajouter au document XML initial une nouvelle signature qui signe le même contenu que la première signature. Le cas le plus simple est quand la signature est détachée. Le contenu signé est donc référencé par une URI externe (voir section 4.2.1). Dans ce cas, il suffit de créer un seul object *XMLSignature* contenant un seul objet *Reference* avec cette URI. La production du document final contenant les deux signature (la signature initiale et la co-signature) ainsi que le contenu signé peut se faire en utilisant les méthodes de la bibliothèque *com.cryptolog.ml* pour coller les différents morceaux.

Dans le cas ou le contenu est présent dans le même document que la signature, la co-signature peut nécessiter plus de travail. Un effort a été fait dans le développement d'une bibliothèque générique de production de signature pour automatiser la co-signature. Cependant, si on fait pas des hypothèses sur le type de la signature d'entrée (la signature à co-signer), la production peut générer un document invalide. Par exemple, si on connaît pas au préalable la position du contenu signé dans le document, on peut invalider la première signature si on insère la co-signature dans un "mauvais" endroit.

6.2.4 Options non implémentées

Plusieurs options définis dans la recommandation W3C[17] n'ont pas été implémentés soit par manque de temps ou absence d'utilité immédiate. Les plus importantes de ces options et qui pourra être intéressant d'ajouter dans les futures versions sont :

- **l'élément RetrievalMethod :** Cet élément permet de désigner un endroit distant où les information de validation de la signature peuvent être trouvées. Cet élément utilise le même modèle des *Reference* en désignant la ressource par une URI. Une utilisation pratique de cet élément est quand un seul document contient plusieurs signature XML qui utilisent une même chaîne de certificats. Pour éviter de donner une copie de cette chaîne dans chaque signature, on peut la transporter dans un unique endroit dans le document et le référencer avec une URI dans des éléments *RetrievalMethod*.
- **l'élément Manifest :** Cet élément est aussi optionnel et si présent il doit être transporté par un élément *Object*(voir section 4.3).
  L'élément *Manifest* peut transporter une liste ou collection de ressources désignées par des éléments *Reference*. Plus précisément, le *Manifest* est en même temps une collection de res-

sources et une ressource puisque lui même il peut être référencé par un élément *Reference*. Point de vue structure, il est comparable à une structure *SignedInfo* sauf qu'il ne contient pas des éléments *SignatureMethod* et *CanonicalizationMethod*.
Cet élément a particulièrement deux utilisations très pratique :
- Quand plusieurs signataires veulent signer une même grande collections de ressources, il est regrettable de devoir désigner cette liste dans l'élément *SignedInfo* de chaque signature. Pour éviter cette redondance, on peut utiliser un élément Manifest pour transporter la collection des ressources puis référencer cet élément par toutes les signatures.
- Le processus de validation de XMLDSIG est définit tels que l'échec de l'étape de validation des références implique l'échec de tout le processus de validation (voir la section 4.2.3). Ce comportement peut ne pas être désirable par certaines applications qui veulent gérer d'une autre manière la validation des éléments *Reference* (par exemple une application peut se limiter à un nombre de références minimal valides pour passer à la deuxième étape de la validation de la signature). Cela peut se réaliser en utilisant un *Manifest*. En effet, dans ce cas la seule *Reference* à valider est celle qui désigne la structure XML du *Manifest*. La validation de l'ensemble des *Reference* transportés devient une responsabilité de l'application et non plus du processus de validation définit dans le standard XMLDSIG.

6.3 XAdES

6.3.1 Solutions existantes

La spécification ETSI de XAdES[9] est relativement récente (dernière version en 2006). Pour W3C, XAdES est encore dans le stade d'une note et non pas une recommandation. Cependant, il existe comme même plusieurs implémentations (le plus souvent partielles) de XAdES. On cite principalement les implémentations de l'IAIK et Baltimore ainsi qu'une implémentation libre dans le cadre du projet OpenXades[††]. Ces implémentations ainsi que d'autres seront abordées dans le chapitre des tests d'intéropérabilité (voir chapitre 7).

6.3.2 Solution implémentée

XAdES étant une extension de XMLDSIG, son API utilise naturellement l'API précédente. Cette utilisation peut se faire de deux façons différentes :

1. la première façon consiste à implémenter une classe *XAdESSignature* qui hérite de la classe *XMLSignature* (représentant une signature XML) en ajoutant les propriétés et les traitements ajoutés par XAdES. Cette solution exige que la lecture d'un document XML doit dans tous les cas donner des instances de *XAdESSignature* (puisqu'une instance de *XMLSignature* ne peut pas être traité comme une signature XAdES).
2. la deuxième solution consiste à utiliser le faite que XAdES est une extension de XMLDSIG dont toutes les information sont transportés dans un élément XML *Object*. Une signature XAdES est donc représentée, comme toute autre signature XML, par une instance *XMLSignature* dont la méthode *getAttachedObject()* permet de récupérer les information spécifiques à XAdES. Indépendemment des formats des signatures contenues dans un document XML (XAdES ou autre format compatible avec XMLDSIG), la lecture donne des instances de XMLSignature. C'est la solution choisie et implémentée.

L'API XAdES est divisé en deux parties :

[††]Projet open source : www.openxades.com

1. Un ensemble de classe représentant les propriétés définis par XAdES[32].

2. Des classes de fabrication (ou Builder) permettant de construire une signature XAdES d'une certaine forme en ajoutant les propriétés appropriées à une signature XMLDSIG.

Les figures 6.3 et 6.4 donnent les diagrammes de classe artiels des packages com.cryptolog.xades et com.cryptolog.xades.helper.

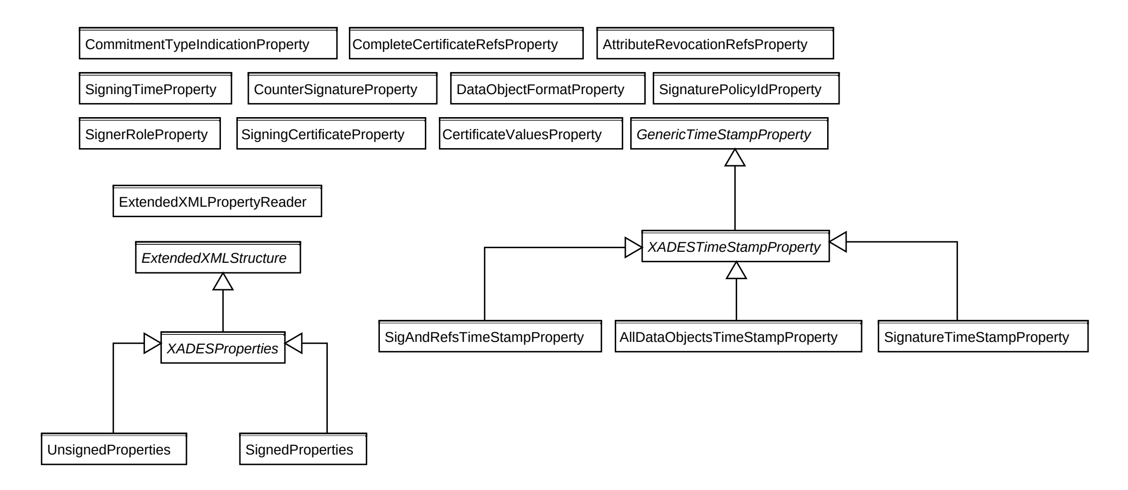


Fig. 6.3 – Diagramme de classe du package xades.

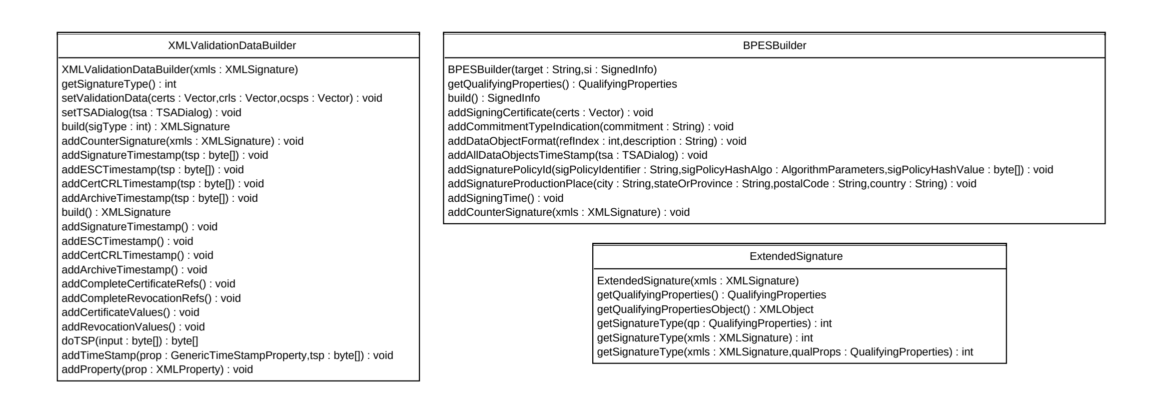


Fig. 6.4 – Diagramme de classe du package xades.helper.

Les propriétés XAdES :

XAdES[32] ajoute un ensemble de propriétés à XMLDSIG. La première partie de l'implémentation de XAdES consistait donc à implémenter ces différentes propriétés. Comme pour XMLDSIG, à chaque élément XML représentant une propriété correspond une classe. Chacune de ces classes héritent de la classe de base *XMLProperty* définissant une propriété XML générique. Une propriété XML peut être définit par :

– Un nom de balise de début qui est le nom de la propriété (exemple : *SigningCertificate*).

- Une URI définissant l'espace des noms pour lequel appartient la propriété.
- Un contenu XML aléatoire.

Une propriété peut être identifiée de façon unique avec le couple : nom de balise et espace des noms. Pour faire la correspondance avec les attributs CMS, chaque propriété peut également être identifié par l'OID de l'attribut CMS correspondant (qui transporte la même information).

D'autre part, la classe *ExtendedXMLProperty* qui hérite de la classe de base *XMLPropertyReader* permet de décoder l'ensemble des propriétés XAdES.

Construction d'une signature XAdES :

XAdES définit deux types de propriétés : les propriétés signées et les propriétés non signées. Les premières doivent obligatoirement être ajoutés par le signataire avant le calcul de la valeur de la signature. Les autres peuvent être ajoutés à n'importe quel moment du cycle de vie de la signature par le signataire ou le vérificateur. On retrouve cette différence dans les classes de construction de XAdES.

- **Construction de XAdES-BES ou XAdES-EPES** : A part la propriété *CounterSignature*, ces deux formes définissent exclusivement des propriétés signées (voir les sections 5.2.1 et 5.2.2). Ces propriétés doivent être attachées à l'objet *SignedInfo* avant d'imposer la signature. Une classe de construction *BEPESBuilder* a été implémentée et fournit des méthodes pour ajouter les différentes propriétés BES ou EPES. Cette classe peut être instancié avec un *SignedInfo* classique. A l'issue de l'ajout, l'appel à une méthode *build()* retourne un nouveau objet *SignedInfo* contenant un élément *Reference* spécifique référençant l'élément *QualifyingProperties* transportant les propriétés ajoutées. Ce *SignedInfo* peut ensuite être traité comme un *SignedInfo* ordinaire (de XMLDSIG). Il peut donc être signé par un simple appel à *sign()*.

- **Construction des autres formes** : Les autres formes T, C, X, X-L et A (voir chapitre 5) ajoutent seulement des propriétés non signés et qui peuvent être ajoutés par le vérificateur et/ou le signataire. Pour ces formes une classe de construction : *DataValidationBuilder* a été implémentée. Cette classe peut être instanciée avec un objet *XMLSignature*, c'est à dire après la production de la valeur de la signature. Cet *XMLSignature* peut représenter une nouvelle signature ou une signature qui vient d'être lu d'un document XML. Elle fournit des méthodes pour l'ajout des différents propriétés. Contrairement à BES et EPES, les propriétés des données de validation peuvent être dépendantes. Par exemple, la propriété *ArchiveTimeStamp* dépend de toutes les propriétés XAdES ajoutées précédemment (y compris les anciennes *ArchiveTimeStamp*) puisque elles contribuent au contenu à dater. Pour cela, la classe de construction doit ajouter les propriétés en temps réel (immédiatement à l'ajout). L'appel à la méthode *build()* retourne un nouveau objet *XMLSignature* dont l'élément *QualifyingProperties* a été crée ou mis à jour. Comme pour un *XMLSignature* classique (de XMLDSIG) la représentation XML du document de la signature peut être obtenue en appelant la méthode *marshal()*.

Validation d'une signature XAdES

La validation d' une signature XAdES peut se faire en deux phases principales. Premièrement, il faut valider la signature en tant que signature XMLDSIG (valider les références signée et la valeur de la signature). Ensuite, il faut valider l'ensemble des propriétés transportées dans la signature. Si la signature est au moins EPES (voir 5.2.2), ces deux étapes peuvent être paramétrés par une politique de validation de signature. La validation et l'interprétation des différentes propriétés peuvent dépendre de l'application et ne sont pas nécessairement spécifié dans la spécification de XAdES[9].

La classe *ExtendedSignature* offre des méthodes utiles pour calculer la forme de la signature et en extraire les propriétés pour être traitées à l'extérieur.

Chapitre 7

Tests d'Intéropérabilité

7.1 Introduction

Les bibliothèques développées implémentent des standards techniques largement utilisés. La conformité des implémentations à ces standards permet de garantir l'intéropérabilité avec d'autres implémentations. Dans ce chapitre, on va décrire un ensemble de tests permettant de vérifier cette intéropérabilité.

Dans la suite on va utiliser le terme "implémentation Cryptolog" pour désigner l'ensemble des bibliothèques implémentées au cours de ce stage, et les termes "autres implémentations" ou "implémentations extérieures" pour désigner des implémentations libres ou développés par d'autres entreprises.

7.1.1 Déroulement des tests

Les tests sont basés sur un ensemble de vecteurs de tests. La plupart de ces fichiers de tests ont été récupérés de Apache XML Security[*]. D'autres fichiers de tests ont été récupérés de l'ECOM PLUG TEST PROJECT [†].

7.1.2 Hypothèses de base

Version des standards

Toutes les implémentations utilisées sont déclarées capable de supporter les dernières versions de XMLDSIG (RFC 3275) et/ou de XAdES (ETSI 101 903 v1.3.2). Sauf quand c'est indiqué, tous les fichiers de signature testés utilisent ces versions.

Conditions de validité

Un test de validation de signature sera dit "réussite" si les conditions suivantes sont toutes vérifiées :

- aucune exception de validité syntaxique n'est signalée.
- la signature est déclaré cryptographiquement valide (selon le processus de validation de base de XMLDSIG). Autrement dit, la valeur de la signature ainsi que les valeurs des hachés sont valides.

[*] Apache XML Security :http ://xml.apache.org/security/dist/java-library/

[†] ECOM PLUG TEST PROJECT : http ://www.ecom.jp/LongTermStrage/en/XAdES_e.html

– la forme XAdES retournée est bien la forme correspondante aux propriétés présentes dans la signature.

Si au moins une des conditions précédentes n'est pas vérifiée, le test sera dit "échec".

Au moment de la réalisation de ces tests, l'entreprise ne dispose pas de bibliothèques complètes pour la vérification des données de validation (certificats, CRLs, OCSP...) d'une signature. Pour cette raison, aucune validation de ce genre ne sera faite. En plus, tous les fichiers de signatures qui vont être utilisés, contiennent explicitement la clé publique du signataire et/ou son certificat. Aucun test ne va porter sur la validité de ces données.

7.1.3 Implémentations extérieures

En plus de l'implémentation Cryptolog, des fichiers de tests issus des implémentations suivantes vont être utilisés :

Apache

- **Nom du produit** : Apache-XML-Security.
- **Description** : Implémente les deux recommandations W3C : signature[17] et chiffrement[35] XML.
- **Platforme** : java et c++.
- **Licence** : Apache Software.
- **Dernière version** : 1.4.1 (Mai 2007).

Baltimore

- **Nom du produit** : Baltimore KeyTools XML.
- **Description** : implémente les deux recommandations W3C : signature[17] et chiffrement[35] XML.
- **Platforme** : java.
- **Licence** : commerciale.

Phaos

- **Nom du produit** : Phaos XML Security suite.
- **Description** : Implémentation des standards : signatures XML[17], chiffrement XML[35], XKMS‡[33] et SAML§[23].
- **Platforme** : java.
- **Licence** : commerciale.

IAIK (Institute for Applied Information Processing and Communication, Graz University of Technology)

- **Nom du produit** : IAIK XML Advanced Electronic Signatures (XAdES) add-on for XML Security Toolkit (XSECT).
- **Description** : signatures XMLDSIG et XAdES.
- **Platforme** : java.
- **Licence** : commerciale.

‡XKMS : XML Key Management Specification
§SAML : Security Assertion Markup Language

– **Dernière version** : 1.3.2

7.2 Matrice des tests

7.2.1 XMLDSIG

Implémentation	Taille du vecteur de tests	Détails couverts par les tests
IAIK	5	– algorithmes : RSA, DSA, HMAC. – transformations : Base64, XPath, Enveloped. – canonicalization : INCLUSIVE avec et sans commentaires. – types de signatures : enveloppante, enveloppé, détaché.
Phaos	6	– algorithmes : RSA, DSA, HMAC. – transformations : Base64, XPath, Enveloped. – canonicalization : INCLUSIVE avec commentaires. – types de signatures : enveloppante, enveloppé.
Baltimore	6	– algorithmes : RSA, DSA. – transformations : XPath, Enveloped. – canonicalization : INCLUSIVE. – types de signatures : enveloppante, enveloppé.
Apache	7	– algorithmes : RSA, DSA. – transformations : Enveloped. – canonicalization : INCLUSIVE. – types de signatures : enveloppante, enveloppé.

7.2.2 XAdES

Implémentation	Taille du vecteur de tests	Version XAdES	Formes XAdES
IAIK	5	1.3.2	BES, EPES, T, C et X-L (type 1).
ECOM	9	1.3.1	BES, T, X-L et A.

7.3 Résultats

7.3.1 XMLDSIG

IAIK

– Tests réussis : 5.
– Tests échoués : 0.

Commentaires :
Tous les fichiers de tests de l'IAIK ont passé la validation. A noter toutefois, que plusieurs fichiers IAIK n'ont pas été sélectionné à cause de certains éléments XMLDSIG non supportés (on cite

XPointer, XMLDSIG XPath) et certains algorithmes de signatures non supportés par Cryptolog (on cite HMAC).

Phaos

- Tests réussis : 5.
- Tests échoués : 1.

Commentaires :
Le seul fichier de signature qui n'a pas passé le test de validation représente une signature XML enveloppante qui signe un contenu XML. Bien que la valeur de la signature a été vérifié correctement, la validation de la seule référence de la signature a échoué. Ceci peut s'expliquer par le faite que l'élément de la référence ne contient aucun algorithme de canonicalization alors que le contenu signé est en XML. D'autre part, certains fichiers du vecteur des signatures Phaos n'ont pas été testé du faite qu'ils contiennent des éléments non encore supportés, à savoir : signature HMAC, la fonction XPath *here*() et la gestion des Manifests. Aussi, certains fichiers qui contiennent des éléments *Manifest* ont été validé sans valider les éléments *Reference* transportés.

Baltimore

- Tests réussis : 6.
- Tests échoués : 0.

Commentaires :
Même remarque que précédemment : a part les signatures qui contiennent des éléments non supportés par l'implémentation Cryptolog, tous les fichiers de tests ont été validé.

Apache

- Tests réussis : 5.
- Tests échoués : 2.

Commentaires :
Les deux fichiers qui n'ont pas passé utilisent des attributs dans le nom d'espace de XML¶[34] avec la canonicalization inclusive[2]. Cela est dû au faite que ces attributs doivent être propagés quand on canonicalize un sous-document d'un document XML, ce qui n'est pas apparemment supporté par XOM[39].

7.3.2 XAdES

IAIK

- Tests réussis : 6.
- Tests échoués : 0.

Commentaires :
Aucun problème particulier à signaler. L'implémentation de l'IAIK produit des signatures totalement conformes à XAdES 1.3.2 qui sont correctement interprétés par l'implémentation Cryptolog.

ECOM

- Tests réussis : 8.
- Tests échoués : 1.

¶XML namespace : http ://www.w3.org/XML/1998/namespace

Commentaires :
Le seul fichier de test qui n'a pas passé transporte les propriétés signés dans un élément *QualifyingPropertiesReference* référencé par un élément *Reference* sans aucun algorithme de canonicalization. Bien que le contenu de cet élément est assez simple et ne permet pas de prévoir qu'est ce qui peut changer s'il est interpété par deux implémentations différentes, cet exemple prouve encore l'importance (et même la nécessité) d'utiliser systématiquement un algorithme de canonicalization dès que le contenu à signer est en XML.

Aussi, notons que le vecteur de tests ECOM utilise la version 1.3.1 de XAdES (la version implémentée est 1.3.2). Les tests ont affirmé qu'il y a une compatibilité ascendante entre les schémas XML des deux versions[‖]. Cependant, il semble que l'interprétation de la propriété ArchiveTimeStamp est différente entre les deux versions. La version 1.3.2 définit implicitement ce qui doit être couvert par ce time-stamp, alors que pour la version 1.3.1 transporte dans la propriété des mécanismes pour référencer explicitement les données protégées.

D'autre part, le vecteur de test pour ECOM est beaucoup plus grand et couvre plus de détails XAdES. Cependant, l'interprétation correcte de ces fichiers de test nécessitait une validation de tous les données (certificats, CRL et OCSP) transportés dans les signature, ce qui n'était pas possible au moment de la réalisation des tests.

[‖] Vue que j'ai pas pu récupéré la version 1.3.1 de la spécification XAdES (seules les versions 1.1.1, 1.2.1, 1.2.2 et 1.3.2 sont disponibles sur le site de l'ETSI), cette affirmation est seulement basée sur les schémas des plus importantes structures trouvées sur le site de l'ECOM (http ://www.ecom.jp/LongTermStrage/en/XAdES01_e.pdf)

Conclusion et perspectives

Nous avons exposé dans ce mémoire l'enjeu important de la sécurité informatique et en particulier la signature numérique comme un futur remplaçant de la signature papier classique (chapitre 3). Nous avons, ensuite, étudié le procédé de production des signatures au format XML (chapitre 4). Avec la la popularisation progressive de la signature numérique , un format évolué pour les signature XML (chapitre 5) a été définit comme une extension à XMLDSIG[17]. Ce format permet de produire en XML des signatures conformes à la directive européenne 1999/93/CE[4] lui donnant une valeur juridique. Les standards des signatures en XML ne présentent aucune innovation en matière de cryptographie. Ils se contentent de puiser les standards existants et qui ont déjà fait leur preuve. Cependant, l'innovation réelle est l'utilisation de XML comme support des données. Ce dernier est devenu le support le plus utilisé pour les échanges des données sur Internet. Son caractère "semi-structuré" a permis de définir des conteneurs pour la plupart des standards de sécurité intervenant dans la gestion d'une architecture à clé publique.

La dernière étape de ce stage consistait à implémenter XMLDSIG et XAdES en Java (chapitre 6). Le développement a été fait en gardant en vue que ces bibliothèques doivent pouvoir être intégrées le plus facilement possible dans des applications existantes de gestion de signatures au format plus classique qui est CMS. Bien que les deux formats convergent en terme d'information cryptographiques transportées, ils divergent en point de vue intention et domaine d'utilisation. En effet, CMS définit des structures "figés" qui sont encodés en binaire selon des règles bien définis et très précises comme DER et BER[1]. Alors que XML est, par nature, un langage de modélisation très ouvert mais aussi très structuré. Son "libéralisme", permet de décrire de plusieurs façons différentes le même contenu XML. Ceci a été le sujet de divers débats sur la pertinence d'utiliser XML comme un support de sécurité et en particulier comme un support pour les signatures numériques dont l'intégrité est l'une des plus importantes exigences. Ce problème a été résolu en partie grâce à des algorithmes de canonicalization. Ces algorithmes permettent de régulariser un contenu XML en définissant la notion de contenus "logiquement équivalents" (chapitre 4). En contre-part, le "libéralisme" de XML offre une grande flexibilité quand à la production des signatures. Selon les besoins, une signature XML peut soit être enveloppée dans la ressource à signer, soit enveloppante en encapsulant le contenu signé. La structuration XML permet aussi la création de toute sorte de hiérarchie entre signatures et contre-signatures dans un même document XML.

L'API développée reste incomplète par rapport aux possibilités offertes par les signatures XML. Le meilleur moyen de continuer le développent de ces API est de développer des modules par profil adaptant XMLDSIG et XAdES à un domaine d'utilisation particulier. Un exemple de profils est "OASIS Web Services Security (WSS)"[29] décrivant l'utilisation de XMLDSIG pour garantir l'intégrité des échanges de messages entre les services web.

Bibliographie

[1] [ASN.1, 04] ASN.1 Information site "Introduction to ASN.1". Disponible sur http ://asn1.elibel.tm.fr/en/.

[2] [C14N, 01] Canonical XML Version 1.0, W3C Recommendation 15 March 2001.

[3] [CRYPTO, 98] Sun Documentation, Introduction to Public-Key Cryptography, http ://docs.sun.com/source/816-6154-10/.

[4] [Directive-Europ, 99] Directive 1999/93/CE du Parlement européen et du Conseil, du 13 décembre 1999, sur un cadre communautaire pour les signatures électroniques.

[5] [DOM, 04] Document Object Model (DOM) Level 3 Core Specification, W3C Recommendation 07 April 2004.

[6] [DSS, 07] Digital Signature Service Core Protocols, Elements, and Bindings, Version 1.0, OASIS Standard.

[7] [ExC14N, 02] Exclusive XML Canonicalization Version 1.0, W3C Recommendation 18 July 2002.

[8] [ETSI TR 102 038, 02] XML format for signature policies. ETSI technical Specification, v1.1.1 Avril 2002.

[9] [ETSI TS 101 903, 06] XML Advanced Electronic Signatures (XAdES). ETSI technical Specification, v1.3.2 March 2006.

[10] [Hash, 95] S.Bakhtiari, R.Safavi and J.Pieprzyk, Cryptography hash functions : A survey.

[11] [PCRYPTO, 76] W. DIFFIE and M. HELLMAN, New directions in cryptography, IEEE Trans. Inform. Theory, 22 (1976), pp. 644-654.

[12] [RFC 791, 81] Internet Protocol, IETF RFC 791.

[13] [RFC 2045, 96] Multipurpose Internet Mail Extensions (MIME) Part One : Format of Internet Message Bodies, IETF RFC 2045

[14] [RFC 2251, 97] Lightweight Directory Access Protocol (v3), IETF RFC 2251.

[15] [RFC 2459, 99] R. Housley, W. Ford, W. Polk, D. Solo, Internet X.509 Public Key Infrastructure : Certificate and CRL Profile, IETF RFC 2459

[16] [RFC 2807, 99] J. Reagle, XML-Signature Requirments, W3C Working Draft 14-October-1999

[17] [RFC 3275, 02] M. Bartel, J. Boyer, B. Fox, B. LaMacchia, E. Simon, XML-Signature Syntax and Processing, W3C recommendation (12 February 2002), IETF RFC.

[18] [RFC 3653, 02] J. Boyer M. Hughes, J. Reagle, XML-Signature XPath Filter 2.0, W3C Recommendation 08 November 2002

[19] [RFC 3852, 04] R. Housley, Cryptographic Message Syntax, IETF RFC, July 2004.

[20] [RFC 3986, 05] Uniform Resource Identifier (URI) : Generic Syntax, IETF RFC 3986.

[21] [RFC 4051, 05] E. Eastlake, Additional XML Security URIs, IETF RFC.

[22] [RFC 4648, 06] The Base16, Base32, and Base64 Data Encodings, IETF RFC 4648.

[23] [SAML, 05] Security Assertion Markup Language (SAML), OASIS Security Services.

[24] [SAX, 01] Simple API for XML,http ://www.saxproject.org/.

[25] [SOAP, 07] Simple Object Access Protocol Version 1.2, W3C Recommendation (Second Edition) 27 April 2007.

[26] [SOAP-SEC, 01] SOAP Security Extensions : Digital Signature, W3C NOTE 06 February 2001.

[27] [UDDI, 02] Universal Description Discovery and Integration, OASIS Published Specification, Dated 19 July 2002.

[28] [WSDL, 01] Web Services Description Language (WSDL) 1.1, W3C Note 15 March 2001.

[29] [WSS, 04] OASIS Web Services Security : SOAP Message Security 1.0. (http ://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf)

[30] [XACML, 05] XML Access Control Markup Language (XACML), OASIS Security Services.

[31] [XACML-DSIG, 03] OASIS XACML XML DSig Profile. Working draft 0.2, 14 March 2003.

[32] [XADES, 03] JC. Cruellas, G. Karlinger, D. Pinkas, J. Ross, XML Advanced Electronic Signatures, W3C Note 20 February 2003.

[33] [XKMS, 01] XML Key Management Specification (XKMS), W3C Note 30 March 2001 .

[34] [XML, 06] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau Extensible Markup Language (XML) 1.0 (Fourth Edition), W3C Recommendation 16 August 2006.

[35] [XML-ENC, 02] XML Encryption Syntax and Processing, W3C Recommendation 10 December 2002.

[36] [XML-NAMES, 06] T. Bray, D. Hollander, A. Layman, R. Tobin, Namespaces in XML 1.0 (Second Edition), W3C Recommendation 16 August 2006

[37] [XML Schema, 06] XML Schema 1.1 Part 1 : Structures, W3C Working Draft 31 August 2006.

[38] [XML-Sec, 02] B. Dournaee, XML Security (chapitre 4 : Introduction to Digital XML Signatures), (http ://www.perfectxml.com/om/XMLSecurity.PDF).

[39] [XOM, 05] XML Object Model, http ://www.xom.nu

[40] [XPath, 07] XML Path Language (XPath) Version 1.0, W3C Recommendation 23 January 2007

[41] [XPointer, 02] XML Pointer Language (XPointer), W3C Working Draft 16 August 2002.

[42] [XSL, 06] Extensible Stylesheet Language (XSL) Version 1.1, W3C Recommendation 05 December 2006.

[43] [XSLT, 99] XSL Transformations (XSLT), W3C Recommendation 16 November 1999.

Lexique

- **ASN.1** : Abstract Syntax Notation number One.
- **BER** : Basic Encoding Rules.
- **CMS** : Cryptographic Message Syntax.
- **CRL** : Certifcate Revocation List.
- **DER** : Distinguished Encoding Rules.
- **DTD** : Document Type Definition.
- **DOM** : Document Object Model.
- **DSA** : Digital Algorithm Signature.
- **DSS** : Digital Signature Standard.
- **ETSI** : European Telecommunications Standards Institute.
- **HTTP** : HyperText Transfer Protocol.
- **IETF** : The Internet Engineering Task Force.
- **IP** : Internet Protocol.
- **MD5** : Message Digest 5.
- **OCSP** : Online Certificate Status Protocol.
- **OID** : Object Identifier.
- **PKCS** : Public-Key Cryptography Standards.
- **RFC** : Request For Comments.
- **SAML** : Security Assertion Markup Language.
- **SAX** : Simple API for XML.
- **SOAP** : Simple object access protocol.
- **SHA1** : Secure Hash Algorithm 1.
- **UDDI** : Universal Description Discovery and Integration.
- **URI** : Uniform Resource Identifier.
- **URN** : Uniform Resource Name.
- **W3C** : World Wide Web Consortium.
- **WSDL** : Web Services Description Language.
- **XADES** : XML ADvanced Electronic Signature.
- **XACML** : XML Access Control Markup Language.
- **XML** : eXtensible Markup Language
- **XMLDSIG** : XML Digital SIGnature.

Exemple d´une signature XADES

La signature XADES-C suivante a été produite par l´implémentation de XADES développé dans le cadre de ce stage. Elle signe deux ressource XML identifiées par :

- "#toSign" : référence un contenu XML attaché au même document XML mais a l´extérieur de la signature.
- "#SignMe" : référence un contenu XML enveloppé dans la signature.

La forme XADES contient les propriétés suivantes :

- SigningTime.
- SigningCertificate.
- SignatureProductionPlace.
- CommitmentTypeIndication.
- SignatureTimeStamp.
- CompleteCertificateRefs.
- CompleteRevocationRefs.


```
<?xml version="1.0" encoding="UTF-8"?>
<crypto:SignatureDocument xmlns:crypto="http://www.cryptolog.com">
  <crypto:SignedData>
    <X Id="toSign" xmlns:t="http://www.test.com"
       xmlns="http://www.testX.com">
      <A>
<B Id="toSign2" xmlns="http://www.test3.com">
Salut </B>
      </A>
    </X>
  </crypto:SignedData>
  <ds:Signature Id="Signature1" xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    <ds:SignedInfo>
      <ds:CanonicalizationMethod
  Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
      <ds:SignatureMethod
  Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
      <ds:Reference URI="#SignMe">
<ds:Transforms>
  <ds:Transform
      Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
</ds:Transforms>
<ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
```

```
<ds:DigestValue>rMK1TAEEdjPSRsOd17KdGATw7qg=</ds:DigestValue>
      </ds:Reference>
      <ds:Reference URI="#toSign2">
<ds:Transforms>
  <ds:Transform Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
</ds:Transforms>
<ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
<ds:DigestValue>pxl+o77f/IxL4MPfvrS2Zomb9gs=</ds:DigestValue>
      </ds:Reference>
      <ds:Reference Type="http://uri.etsi.org/01903#SignedProperties"
    URI="#Signature1_SignedProperties">
<ds:Transforms>
  <ds:Transform Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
</ds:Transforms>
<ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
<ds:DigestValue>N6NaLyulkeAGlO2KO+mVis6mm/M=</ds:DigestValue>
      </ds:Reference>
    </ds:SignedInfo>
    <ds:SignatureValue>WwPSFlGpCWV+IUkqksby9i5Ze6WjsRj/8V3jQj3kuB
    U9zdy5xpOCnHvJT5WdF/jywBu4/vC7qORcjxpBce7OWA==</ds:SignatureValue>
    <ds:KeyInfo>
      <ds:KeyValue>
<ds:RSAKeyValue>
  <ds:Modulus>uKBSyP8JxARZ4BMb2UyHEJXQmka8oVbD2abc/5e+SfK
  twOCYGKOmFC6dtFt7XKkz38piDS9Or3j/GPbip40kPw==</ds:Modulus>
  <ds:Exponent>AQAB</ds:Exponent>
</ds:RSAKeyValue>
      </ds:KeyValue>
    </ds:KeyInfo>
    <ds:Object>
      <database Id="SignMe" name="infos">
<table name="codeDepartement">
  <record>
    <field name="ID" type="string">01</field>
    <field name="NOM" type="string">Ain</field>
  </record>
  <record>
    <field name="ID" type="string">02</field>
    <field name="NOM" type="string">Aisne</field>
  </record>
  <record>
    <field name="ID" type="string">03</field>
    <field name="NOM" type="string">Allier</field>
  </record>
</table>
      </database>
    </ds:Object>
    <ds:Object>
      <xades:QualifyingProperties Target="#Signature1"
  xmlns:xades="http://uri.etsi.org/01903/v1.3.2#">
```

```
<xades:SignedProperties
    Id="Signature1_SignedProperties">
  <xades:SignedSignatureProperties>
    <xades:SigningTime>2007-04-31T10:43:56.154Z</xades:SigningTime>
    <xades:SigningCertificate>
      <xades:Cert>
<xades:CertDigest>
  <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
  <ds:DigestValue>0jtCwg0UJVySFMZWC9f0EWZv/H0=</ds:DigestValue>
</xades:CertDigest>
<xades:IssuerSerial>
  <xades:X509IssuerName>EMAILADDRESS=thomas.pornin@cryptolog.com,CN=Test
  CA,O=Cryptolog,L=Paris,ST=Paris,C=FR</xades:X509IssuerName>
  <xades:X509SerialNumber>50</xades:X509SerialNumber>
</xades:IssuerSerial>
      </xades:Cert>
    </xades:SigningCertificate>
    <xades:SignatureProductionPlace>
      <xades:City>Paris</xades:City>
      <xades:StateOrProvince>Creteil</xades:StateOrProvince>
      <xades:PostalCode>94000</xades:PostalCode>
      <xades:CountryName>France</xades:CountryName>
    </xades:SignatureProductionPlace>
  </xades:SignedSignatureProperties>
  <xades:SignedDataObjectProperties>
    <xades:CommitmentTypeIndication>
      <xades:CommitmentTypeId>
<xades:Identifier Qualifier="OIDAsURI">Tester</xades:Identifier>
      </xades:CommitmentTypeId>
      <xades:AllSignedDataObjects/>
    </xades:CommitmentTypeIndication>
  </xades:SignedDataObjectProperties>
</xades:SignedProperties>
<xades:UnsignedProperties>
  <xades:UnsignedSignatureProperties>
    <xades:SignatureTimeStamp>
      <ds:CanonicalizationMethod
  Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
      <xades:EncapsulatedTimeStamp>...</xades:EncapsulatedTimeStamp>
    </xades:SignatureTimeStamp>
    <xades:CompleteCertificateRefs>
      <xades:CertRefs>
<xades:Cert>
  <xades:CertDigest>
    <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
    <ds:DigestValue>0jtCwg0UJVySFMZWC9f0EWZv/H0=</ds:DigestValue>
  </xades:CertDigest>
  <xades:IssuerSerial>
    <xades:X509IssuerName>EMAILADDRESS=thomas.pornin@cryptolog.com,CN=Test
    CA,O=Cryptolog,L=Paris,ST=Paris,C=FR</xades:X509IssuerName>
```

```
    <xades:X509SerialNumber>50</xades:X509SerialNumber>
  </xades:IssuerSerial>
</xades:Cert>
      </xades:CertRefs>
    </xades:CompleteCertificateRefs>
    <xades:CompleteRevocationRefs>
      <xades:CRLRefs>
<xades:CRLRef>
  <xades:DigestAlgAndValue>
    <xades:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
    <xades:DigestValue>ZsniYJSP5NOayv0pV6i3xd2c1Cw=</xades:DigestValue>
  </xades:DigestAlgAndValue>
  <xades:CRLIdentifier>
    <xades:Issuer>EMAILADDRESS=ca-officer@selso.com,CN=Selso
    CA,O=Selso,L=Paris,C=FR</xades:Issuer>
    <xades:IssueTime>2007-01-11T07:26:19.0Z</xades:IssueTime>
  </xades:CRLIdentifier>
</xades:CRLRef>
      </xades:CRLRefs>
    </xades:CompleteRevocationRefs>
  </xades:UnsignedSignatureProperties>
</xades:UnsignedProperties>
      </xades:QualifyingProperties>
    </ds:Object>
  </ds:Signature>
</crypto:SignatureDocument>
```

Schémas des signatures XML

Ce schémas n'est pas complets et se limite aux structures les importantes. Le schémas complet peut être trouvé ici : http ://www.w3.org/TR/xmldsig-core/xmldsig-core-schema.xsd

```
<?xml version="1.0" encoding="utf-8"?>

<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
        targetNamespace="http://www.w3.org/2000/09/xmldsig#"
        version="0.1" elementFormDefault="qualified">

<!-- Start Signature -->

<element name="Signature" type="ds:SignatureType"/>
<complexType name="SignatureType">
  <sequence>
    <element ref="ds:SignedInfo"/>
    <element ref="ds:SignatureValue"/>
    <element ref="ds:KeyInfo" minOccurs="0"/>
    <element ref="ds:Object" minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
  <attribute name="Id" type="ID" use="optional"/>
</complexType>

  <element name="SignatureValue" type="ds:SignatureValueType"/>
  <complexType name="SignatureValueType">
    <simpleContent>
      <extension base="base64Binary">
        <attribute name="Id" type="ID" use="optional"/>
      </extension>
    </simpleContent>
  </complexType>

<!-- Start SignedInfo -->

<element name="SignedInfo" type="ds:SignedInfoType"/>
<complexType name="SignedInfoType">
  <sequence>
    <element ref="ds:CanonicalizationMethod"/>
    <element ref="ds:SignatureMethod"/>
```

```
    <element ref="ds:Reference" maxOccurs="unbounded"/>
  </sequence>
  <attribute name="Id" type="ID" use="optional"/>
</complexType>

<!-- Start Reference -->

<element name="Reference" type="ds:ReferenceType"/>
<complexType name="ReferenceType">
  <sequence>
    <element ref="ds:Transforms" minOccurs="0"/>
    <element ref="ds:DigestMethod"/>
    <element ref="ds:DigestValue"/>
  </sequence>
  <attribute name="Id" type="ID" use="optional"/>
  <attribute name="URI" type="anyURI" use="optional"/>
  <attribute name="Type" type="anyURI" use="optional"/>
</complexType>

  <element name="Transforms" type="ds:TransformsType"/>
  <complexType name="TransformsType">
    <sequence>
      <element ref="ds:Transform" maxOccurs="unbounded"/>
    </sequence>
  </complexType>

<!-- End Reference -->

<element name="DigestMethod" type="ds:DigestMethodType"/>
<complexType name="DigestMethodType" mixed="true">
  <sequence>
    <any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
  <attribute name="Algorithm" type="anyURI" use="required"/>
</complexType>

<!-- End SignedInfo -->

<!-- Start KeyInfo -->

<element name="KeyInfo" type="ds:KeyInfoType"/>
<complexType name="KeyInfoType" mixed="true">
  <choice maxOccurs="unbounded">
    <element ref="ds:KeyName"/>
    <element ref="ds:KeyValue"/>
    <element ref="ds:RetrievalMethod"/>
    <element ref="ds:X509Data"/>
    <element ref="ds:PGPData"/>
    <element ref="ds:SPKIData"/>
    <element ref="ds:MgmtData"/>
```

```
    <any processContents="lax" namespace="##other"/>
    <!-- (1,1) elements from (0,unbounded) namespaces -->
  </choice>
  <attribute name="Id" type="ID" use="optional"/>
</complexType>

  <element name="KeyName" type="string"/>
  <element name="MgmtData" type="string"/>

  <element name="KeyValue" type="ds:KeyValueType"/>
  <complexType name="KeyValueType" mixed="true">
   <choice>
     <element ref="ds:DSAKeyValue"/>
     <element ref="ds:RSAKeyValue"/>
     <any namespace="##other" processContents="lax"/>
   </choice>
  </complexType>

  <element name="RetrievalMethod" type="ds:RetrievalMethodType"/>
  <complexType name="RetrievalMethodType">
    <sequence>
      <element ref="ds:Transforms" minOccurs="0"/>
    </sequence>
    <attribute name="URI" type="anyURI"/>
    <attribute name="Type" type="anyURI" use="optional"/>
  </complexType>

<!-- Start X509Data -->

<element name="X509Data" type="ds:X509DataType"/>
<complexType name="X509DataType">
  <sequence maxOccurs="unbounded">
    <choice>
      <element name="X509IssuerSerial" type="ds:X509IssuerSerialType"/>
      <element name="X509SKI" type="base64Binary"/>
      <element name="X509SubjectName" type="string"/>
      <element name="X509Certificate" type="base64Binary"/>
      <element name="X509CRL" type="base64Binary"/>
      <any namespace="##other" processContents="lax"/>
    </choice>
  </sequence>
</complexType>

<complexType name="X509IssuerSerialType">
  <sequence>
    <element name="X509IssuerName" type="string"/>
    <element name="X509SerialNumber" type="integer"/>
  </sequence>
</complexType>
```

```
<!-- End X509Data -->

<!-- End KeyInfo -->

<!-- Start Object (Manifest, SignatureProperty) -->

<element name="Object" type="ds:ObjectType"/>
<complexType name="ObjectType" mixed="true">
  <sequence minOccurs="0" maxOccurs="unbounded">
    <any namespace="##any" processContents="lax"/>
  </sequence>
  <attribute name="Id" type="ID" use="optional"/>
  <attribute name="MimeType" type="string" use="optional"/> <!-- add a grep facet -->
  <attribute name="Encoding" type="anyURI" use="optional"/>
</complexType>

<element name="Manifest" type="ds:ManifestType"/>
<complexType name="ManifestType">
  <sequence>
    <element ref="ds:Reference" maxOccurs="unbounded"/>
  </sequence>
  <attribute name="Id" type="ID" use="optional"/>
</complexType>

<element name="SignatureProperties" type="ds:SignaturePropertiesType"/>
<complexType name="SignaturePropertiesType">
  <sequence>
    <element ref="ds:SignatureProperty" maxOccurs="unbounded"/>
  </sequence>
  <attribute name="Id" type="ID" use="optional"/>
</complexType>

   <element name="SignatureProperty" type="ds:SignaturePropertyType"/>
   <complexType name="SignaturePropertyType" mixed="true">
     <choice maxOccurs="unbounded">
       <any namespace="##other" processContents="lax"/>
       <!-- (1,1) elements from (1,unbounded) namespaces -->
     </choice>
     <attribute name="Target" type="anyURI" use="required"/>
     <attribute name="Id" type="ID" use="optional"/>
   </complexType>

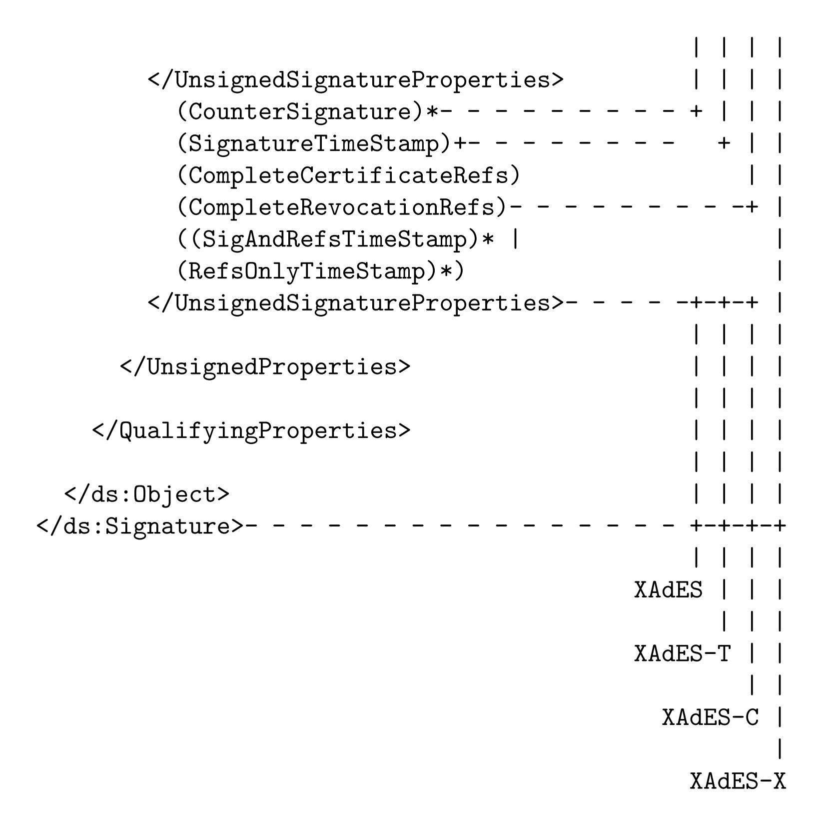
<!-- End Object (Manifest, SignatureProperty) -->

<!-- End Signature -->

</schema>
```

Schémas des signatures XADES

```
         XMLDISG
                                   |
<ds:Signature ID?>- - - - - - - - +- - - - - - +-+-+-+
  <ds:SignedInfo>                  |             | | | |
    <ds:CanonicalizationMethod/>  |             | | | |
    <ds:SignatureMethod/>          |             | | | |
    (<ds:Reference URI? >          |             | | | |
      (<ds:Transforms>)?           |             | | | |
      <ds:DigestMethod>            |             | | | |
      <ds:DigestValue>             |             | | | |
    </ds:Reference>)+              |             | | | |
  </ds:SignedInfo>                 |             | | | |
  <ds:SignatureValue>              |             | | | |
  (<ds:KeyInfo>)? - - - - - - - - +             | | | |
                                                 | | | |
  <ds:Object>                                    | | | |
                                                 | | | |
    <QualifyingProperties>                       | | | |
                                                 | | | |
      <SignedProperties>                         | | | |
                                                 | | | |
        <SignedSignatureProperties>              | | | |
          (SigningTime)                          | | | |
          (SigningCertificate)                   | | | |
          (SignaturePolicyIdentifier)            | | | |
          (SignatureProductionPlace)?            | | | |
          (SignerRole)?                          | | | |
        </SignedSignatureProperties>             | | | |
                                                 | | | |
        <SignedDataObjectProperties>             | | | |
          (DataObjectFormat)*                    | | | |
          (CommitmentTypeIndication)*            | | | |
          (AllDataObjectsTimeStamp)*             | | | |
          (IndividualDataObjectsTimeStamp)*      | | | |
        </SignedDataObjectPropertiesSigned>      | | | |
                                                 | | | |
      </SignedProperties>                        | | | |
                                                 | | | |
      <UnsignedProperties>                       | | | |
```



```
                                                | | | |
        </UnsignedSignatureProperties>          | | | |
          (CounterSignature)*- - - - - - - - - + | | |
          (SignatureTimeStamp)+- - - - - - -   + | |
          (CompleteCertificateRefs)              | |
          (CompleteRevocationRefs)- - - - - - - -+ |
          ((SigAndRefsTimeStamp)* |                |
          (RefsOnlyTimeStamp)*)                    |
        </UnsignedSignatureProperties>- - - - -+-+-+ |
                                                | | | |
      </UnsignedProperties>                     | | | |
                                                | | | |
    </QualifyingProperties>                     | | | |
                                                | | | |
  </ds:Object>                                  | | | |
</ds:Signature>- - - - - - - - - - - - - - - - +-+-+-+
                                                | | | |
                                          XAdES | | |
                                                  | | |
                                          XAdES-T | |
                                                    | |
                                            XAdES-C |
                                                      |
                                              XAdES-X
```

Cahier des charges

Rappel des coordonnées du stage

Cryptolog International SAS 16-18 rue Vulpian F-75013 Paris France

Rappel du sujet :

Etude et implémentation des protocoles XMLDSIG (XML-Signature Syntax and Processing) et XADES (XML Advanced Electronic Signatures) pour les interfacer dans la suite avec des applications mettant en oeuvre des séquestres de clés.

Mots clés

Signature numérique, chiffrement asymétrique, RSA, SHA1, certificat numérique, XML, canonicalization.

Contexte général

Actuellement XML est de plus en plus utilisé comme un moyen de stockage de données structurées sur Internet. Il est également utilisé comme support pour plusieurs protocoles de communication déployés dans les architectures des services web. Le protocole SOAP (Simple Object Access Protocol) en est un exemple. La sécurisation des requêtes XML est devenue donc un besoin. Dans ce cadre deux standards ont été définis :

- Chiffrement : XML Encryption Syntax and processing.
- Signature numérique : XML Signature Syntax and processing (XMLDSIG).

XMLDSIG permet de signer un contenu XML mais aussi tout autre contenu numérique. Il est basé sur l'utilisation de la cryptographie asymétrique à clés publiques. Les spécifications de ce standard permettent de garantir l'authentification du message ainsi que son intégrité. La non répudiation n'est pas directement garantit mais laissée à la responsabilité de l'application.

D'autre part, XMLDSIG permet de définir des transformations à effectuer sur le contenu à signer. Ces transformations permettent par exemple de signer plusieurs parties d'un même document par plusieurs personnes. En particulier XMLDSIG permet de créer trois types de signatures :

- Signatures détachées : Cas standard. Le document XML de la signature est externe a la ressource signée.
- Signatures enveloppées : La signature est contenue dans la ressource signée.
- Signatures enveloppantes : La ressource signée est contenue dans le document XML de la signature.

Example d'une signature enveloppée dans une requête SOAP :

```
<SOAP:Envelope xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/">
...
<SOAP:Body>
<B xmlns:n2="&bar;">
<Signature xmlns="&dsig;">
</SignedInfo>
<Reference URI="">
<Transforms>....</Transforms>
<DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
<DigestValue>....</DigestValue>
</Reference>
</SignedInfo>
<SignatureValue>....</SignatureValue>
</Signature>
<C ID="signme" xmlns="&baz;"/>
</B>
</SOAP:Body>
</SOAP:Envelope>
```

XAdES est une extension de XMLDSIG qui permet de lui ajouter la notion de non-répudiation. XAdES est aussi conforme à la Directive 1999/93/EC du Parlement Européen et du Conseil du 13 décembre 1999 sur le cadre communautaire des signatures électroniques et qui peut donc être utilisée pour arbitrage en cas de litige entre le signataire et le vérificateur.

L'existant

Il existe actuellement plusieurs implémentations Java de XMLDSIG. Les deux les plus importantes :

- JSR (Java Specification Request) 105 par SUN. Cette implémentation fait partie intégrante de l'API stanadrd Java à partir de sa version 1.6.
- Apache XML Project : Les standards de signature et chiffrement XML sont implémentés par le composant XML Security du projet apache.

Bien que les licences de ces deux implémentations sont compatibles avec une utilisation commerciale, plusieurs raisons ont amenées l'entreprise à penser à faire sa propre implémentation. Parmi lesquels :

- La JSR n'est disponible qu'à partir de java 6. Cependant, la majorité des clients de l'entreprise utilisent encore les versions 3 et 4 (Il y a même des clients qui utilisent encore les versions 1 et 2 !)
- Le projet Apache ainsi que la JSR utilisent la JCA (Java Cryptography Architecture) intégré à Java (en partie depuis 1.2). L'entreprise préfère utiliser ses propres bibliothèques cryptographique développées en interne ce qui facilitera l'utilisation de la bibliothèque XMLDSIG par les applications de l'entreprise.

D'autre part, comme toute application basée sur XML, on a besoin de pouvoir générer des documents XML conformes à certains schémas et aussi naviguer dans des structures XML pour en extraire les informations nécessaires. A partir de sa version 5 java fournit une bibliothèque assez complète pour traiter les documents XML. Cependant, pour les raisons précédentes on est amenée a utiliser des bibliothèques externes compatibles avec des versions plus ancienne de Java. Parmi ces bibliothèques, les plus connues sont : JDOM, Xerces, XOM.

Le projet

Le projet consiste donc à implémenter en première partie le standard XML, avant d'étudier l'ajout des fonctionnalités de l'extension XADES. D'autre part, l'entreprise dispose d'une implémentation d'un standard plus ancien pour les signatures digitales : CMS (Cryptographie Message Syntax). CMS définit un format pour signer et chiffrer tout type de données. L'implémentation de XMLDSIG devra être compatible le plus possible avec les interfaces de CMS pour pouvoir s'utiliser de la même manière. La première partie du stage consiste à lire le standard XMLDSIG et étudier les technologies liées. Les documents de base sont les suivants :

- XML Signature Syntax and Processing (RFC 3275).
- XML Signature Requirements (RFC 2807).
- Signature XPATH Filter 2.0 (RFC 3653).
- Additional XML Security URIs (RFC 4051).
- Canonical XML 1.0 (RFC 3076).
- Exclusive XML canonicalization 1.0 (RFC 3741).

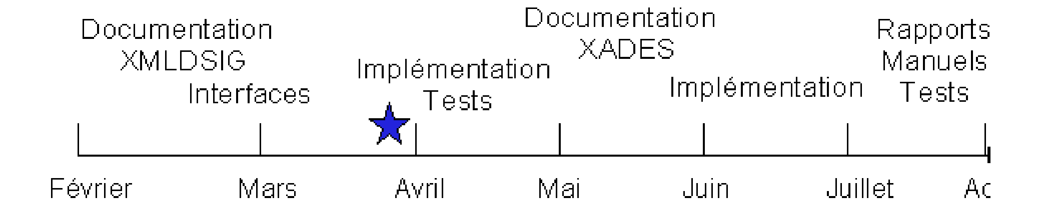
La première tache qui m'est demandée est la rédaction d'un document présentant le standard XMLDSIG, les contextes possibles de son utilisation ainsi que des choix d'implémentation.

Après cette étude, suivra l'écriture d'un ensemble d'interfaces. Ces interfaces constitueront en plus de la RFC de XMLDSIG le cahier des charges de l'implémentation. La bibliothèque doit être conforme le plus possible aux spécifications pour garantir l'interopérabilité. Par exemple, un document signé avec l'implémentation de Java 6 doit pouvoir être validé par la bibliothèque et vice-versa.

D'autre part, l'implémentation ne doit utiliser que les bibliothèques internes de l'entreprise en tout ce qui concerne fonctions cryptographique, hashage, certificat ou signatures. L'utilisation de l'API standard de Java doit se restreindre à ce qui est disponible a partir de java 1.2.

Il y a aussi une contrainte sur la taille de la bibliothèque à développer. Celle-ci va devoir s'intégrer dans une Applet java a taille réduite et qui permet de créer/valider plusieurs types de signatures.

Planning



Development : XMLDSIG API

This page last changed on Jul 26, 2007 by benmbarka.

Introduction :

This document is intended to explain the use of the XMLDSIG API.

The main class of the API is the XMLSignature class which represents an XML signature. An XMLSignature can be obtained by signing a SignedInfo element (which represents the content actually covered by the signature) or by unmarshaling an XML document. The two cases are discussed in the next sections.

XML Signature production :

A typical XML signature is composed of two mandatory elements : the SignedInfo and the SignatureValue, and one optional element : the KeyInfo.

Method 1 :

The API can produce an XML signature in four main steps : (The full code : SimpleGenerate.java)

1. First of all, you must create the Reference addressing the data object to sign with an URI. Lets say you want to sign only one file.
    a. Create the Reference :

```
Reference ref = new Reference("#toSign", null, null, Algo.SHA1);
```

    We assume that the file to sign is not an XML file, so no need to add any transform algorithm. The SHA1 algorithm will be used to digest the signed content.

    b. You need now to attach the content to sign. Suppose you want to create an enveloping signature (the content of the file will be base64 encoded and put into the signature)

```
ref.attachContent(new ContentSource(new FileInputStream("FileToSign"), XMLSignature.ENVELOPING,
false));
```

2. Now you can create the SignedInfo. This element represents the content that will be covered by the signature value. In addition to the list of the Reference addressing the signed data objects, it contains the signature and the canonicalization algorithm identifiers. We will use the EXCLUSIVE canonicalization algorithm and RSA for the signature :

```
Vector refs = new Vector();
refs.add(ref);
SignedInfo si = new SignedInfo(
                        new CanonicalizationMethod(TransformFactory.EXC_C14N),
                        new SignatureMethod(Algo.RSASignature_PKCS1_SHA1),
                        refs);
```

3. Now you have just to sign the created SignedInfo to obtain the XMLSignature. The signer can send into the signature any information (DN, certificates...) which can be used by the verifier to select its public key. For example, he can add explicitly its public key.

```
XMLSignature xmls = si.sign(new KeyInfo(null, new KeyValue(publicKey), null),
signatureGenerator);
```

4. The obtained XML signature is ready to be marshaled to obtain the XML representation of the signature. Before the marshaling, you can attach unsigned signature properties or any arbitrary XML data :

```
xmls.addUnsignedProperty(property);
xmls.marshal(System.out);
```

Method 2 :

The XMLSigUtils class provides helpful methods to quickly create a SignedInfo ready to be signed :

```
SignedInfoTemplate sit = XMLSigUtils.newSignedInfoTemplate("#toSign", null, false,
CERTS[0].encode(), Algo.SHA1,
                                        XMLSignature.ENVELOPING, Algo.RSASignature_PKCS1_SHA1,
TransformFactory.EXC_C14N);
SignedInfo si = sit.getSignedInfo();
```

It also provides methods to create a SignedInfo based only on the hash of the data to sign :

```
SignedInfoTemplate sit = XMLSigUtils.newSignedInfoTemplate("#toSign", null, false,
CERTS[0].encode(), Algo.SHA1,
                                        null, Algo.RSASignature_PKCS1_SHA1,
TransformFactory.EXC_C14N);
SignedInfo si = sit.getSignedInfo();
```

Countersignatures :

A countersignature is a signature which signs the SignatureValue element of the encapsulated signature. It can be produced in two main steps :

1. First, get the appropriate SignedInfo containing one Reference addressing the SignatureValue of the signature to countersign :

```
SignedInfo counterSignedInfo = xmls.countersign(TransformFactory.EXC_C14N,
Algo.RSASignature_PKCS1_SHA1);
```

2. Then you can sign the SignedInfo using any of the sign method to get the XMLSignature.

```
XMLSignature sountersig = counterSignedInfo.sign(new KeyInfo(null, new KeyValue(publicKey),
null), signatureGenerator);
```

The following code gives an example of how to countersign an existing XML signature and add the countersignature as a XADES CounterSignature property.

Co-signatures :

A co-signature is a signature which signs the same data object(s) as the existing signature to co-sign. The production of a co-signature may be complicated because it depends on the template of the signature to co-sign. I made an attempt to make the production generic using subclass of an XMLSignedData class (libsiggen-xml) which work properly for signatures produced by our XMLDSIG implementation but not surely for an external signature. The way to encode the co-signature and the co-signed signature in the same XML document depends on the template of the signature (ENVELOPED, ENVELOPING or ATTACHED). Thus, the user must get this information from the existing signature before encoding the co-signature.

A simpler way, is to create the co-signature externally or to create builders when specific profiles are used. The XMLSignature class provides a coSign method which returns an appropriate SignedInfo for a co-signature.

The following sample code gives an example of how to use this method to create a co-signature and encode it within the same XML document as the co-signed signature.

XML Signature verification :

The XMLDsigFactory class provides multitude of methods to unmarshal an XML content. Any XML signature found, is represented by an XMLSignature object.

The XMLSignature class provides many method to verify either the signature value, the Reference(s) digest value(s) or both. If the signed content is external (not included within the same XML document as the signature), it must be attached to the corresponding Reference before the validation. Even if the content is included within the XML document (ENVELOPED, ENVELOPING or ATTACHED), an external content can be attached to override the default dereferencing process.

The verification of any validation data (certificates, CRLs...) must be done externally, before calling the verify method with the public key on the XMLSignature.

Signature verification sample code.

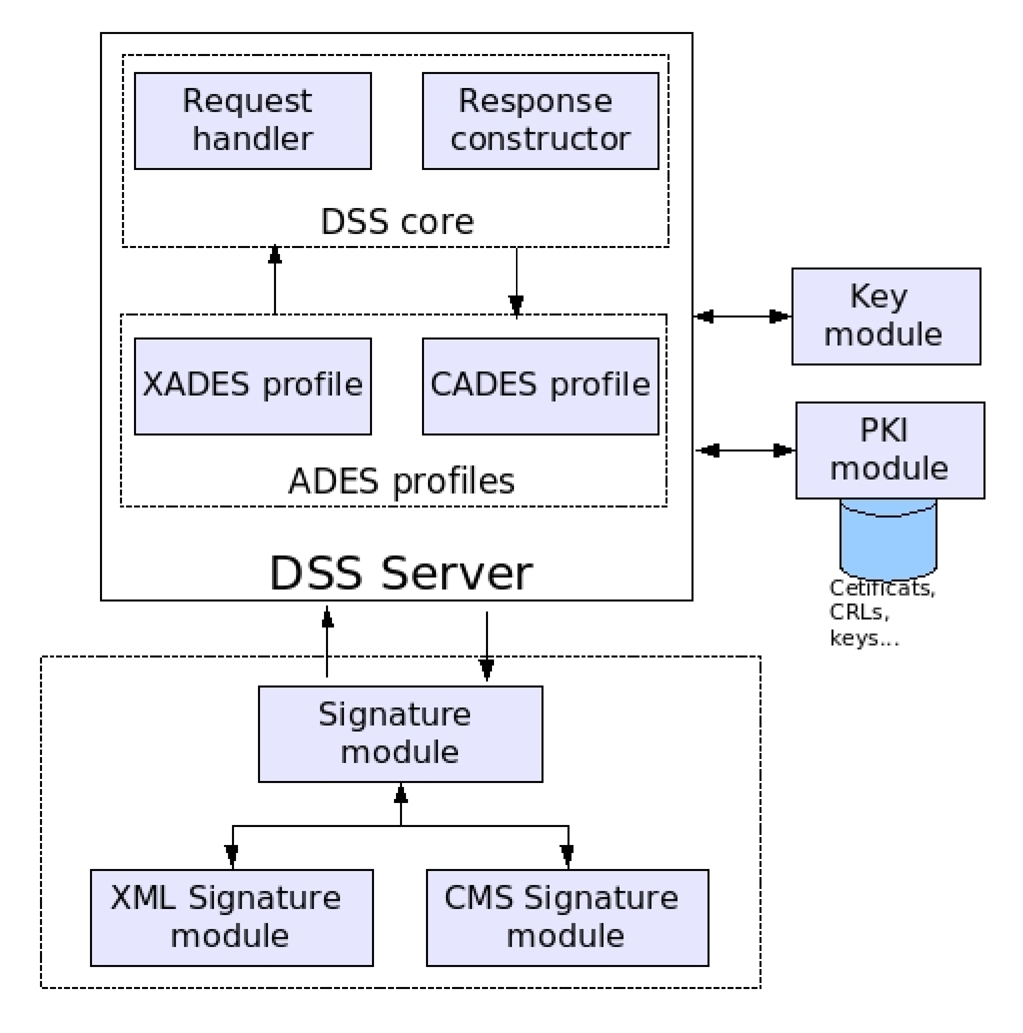
This page last changed on Jul 05, 2007 by benmbarka.

DSS Server Architecture

The aim of the project is to develop a DSS server supporting XML and CMS formats. In addition to the DSS core protocols, it must also support the ADES profiles. Such server should be able to :

- create an XML or CMS signature.
- create a XADES or CADES signature in any advanced predefined form.
- validate any signature form mentioned above and ,when requested, update the given signature by incorporating additional unsigned attributes.

The basic server architecture :



The server must handle two type of queries : SignRequest and VerifyRequest. These requests must be received from an authenticated user using a secure channel. The DSS paper defines the use of HTTP or SOAP as transport protocols and TLS to manage the security (confidentiality and integrity) of the communication channel between the client and the server.

Handling the SignRequest :

The server may handle an authenticated SignRequest in the following steps :

1. decodes the client request.
2. checks that it is a valid request and that the requested profile and options are supported by the server. If it determines that the request is invalid or it contains unsupported options that can not be ignored, it returns a SignResponse containing only a failure code.
3. gets the document to sign.
4. computes the signature over the document in the appropriate format. The server may add additional signed or unsigned attributes if required by the client or implied by the server policy.
5. builds a SignResponse.
6. if the creation process succeeded, adds the signature, any optional output required by the client and a SUCCESS code to the SignResponse.
7. otherwise, adds a FAILURE code to the SignResponse.
8. returns the response to the client.

The server may parametrize the signature creation (step **4**) using :

- the selected profile.

- any optional input provided into the SignRequest. In particular :
  - the ServicePolicy option which may indicate a specific signature policy.
  - the ClaimedIdentity option which may indicate the identity of the client.

Also, for the step **4**, the server needs, in addition to the document to sign, the signer private key. This key may be retrieved using :

- the user profile used to authenticate the client.
- the ClaimedIdentity option if present.
- the KeySelector option if present. The client can use this option to give indications (identifiers, certificates...) about the key to use.
- The user session carried from the authentication layer.

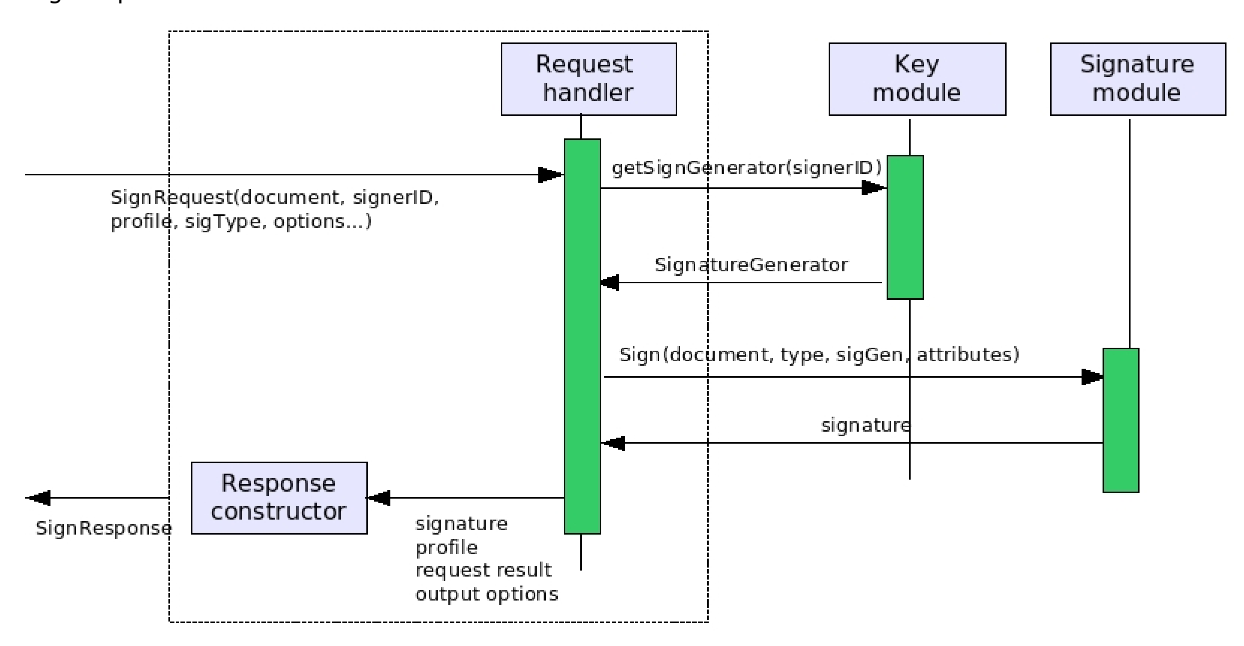
Thus, the server needs to communicate with a Key module (RemoteToken ?) to get the private key (a SignatureGenerator ?). When, the authenticated user has permissions to use a set of private keys, he should select a specific key using the KeySelector option which must provide an identifier mapped to a unique key.

When an advanced signature is required (XADES or CADES), the server may need to produce the signature under a specific signature policy. The server can either :

- use always the same signature policy .
- or manage a store of signature policies and allow the client to select a specific one using the ServicePolicy option .

The form (the incorporated properties) of the produced signature depends on three parameters. A property is added to the signature if it is implied by at least one of the following parameters :

1. The selected signature policy (unsigned and signed mandated properties).
2. The Properties option if present. The values carried into this option may be ignored if already implied by the signature policy.
3. The SignatureForm option. This option is used only if it implies properties which are not already added by the two previous parameters.
   The following figure summarizes the interactions between the server modules to handle a SignRequest :



Handling the SignRequest options :

The ServicePolicy option :

This option indicates a specific signature policy to use. This option is mandatory, if the server can support multiple policies .

The ClaimedIdentity option :

This option indicate the identity of client making the request .

The SignatureType option :

Indicates the type of the signature (CMS or XML). This option is mandatory.

The SignatureForm option :

This option indicates the form of the signature if a XADES or CADES signature is required. The required signature form must be compatible with the selected signature policy (For instance, if the signature policy implies that the SigningCertificate property must be present, the required form must be at least XADES-T

(or CADES-T)).

The Properties option :

This option indicates a set of signed and unsigned properties to add to the signature being created. The properties are indicated using URIs. The client can also provied explicitly the properties values.
For the signed properties, The server must check that the provided values correspond with the used signature policy. Also, the server may add additional properties implied by the signature policy.

The KeySelector option :

The server may use this option to select the user signing key. This option is useful if an authenticated user is allowed to use more than one private key.

The AddTimestamp option :

If this option is present, the server must add a SignatureTimestamp property to the signature before returning the SignResponse.

The IncludeObject (for XML signature) and IncludeEcontent (for CMS signatures) options :

These options indicate to the server if the data object to sign must be included into the signature.

Handling the VerifyRequest :

A VerifyRequest can be handled as follows :

1. decodes the request.
2. checks that it is a valid request and that the requested profile and options are supported by the server. If it determines that the request is invalid or it contains unsupported options that can not be ignored, it returns a VerifyResponse containing only a failure code.
3. gets the signer public key using information given into the signature or into the request.
4. checks the validity of the signature.
5. If required, updates the signature (by time-stamping the signature value or adding other unsigned attributes).
6. builds a VerifyResponse with the validation result, any information required by the client and optionally the updated signature.
7. returns the response to the client.

Like for the SignRequest, the server may parameterize its verification process using the selected profile, the ServicePolicy option and the ClaimedIdentity option if present.

To get the signer certificate (step 3) and validate it, the server can use the validation data provided into the signature and other input options. In particular, the client can provide additional validation data (certificate chain, CRLs...) in an AdditionalKeyInfo option.

The figure bellow summarizes the interactions between the server modules to handle a VerifyRequest :

Server configuration :

To parameterize the server process, it may be configured. The main configuration parameters :

- The supported signature formats.
- The supported DSS profiles.
- The supported input options.
- The default signature policy and additional supported policies.
- Whether or not to support multiple signature verification.
- ...
  Each profile may have an additional configuration file to specify specific supported options and signature policies.